University of Cape Town

Masters in Engineering Thesis

An Empirical Assessment Of Real-Time Progressive Stereo Reconstruction

Author:
Matthew Westaway

Supervisor:
Dr. George Sithole

A thesis proposal submitted in fulfilment of the requirements for the degree of Masters of Engineering

November 2015

Declaration of Authorship

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own

Signed: Signed by candidate

Date:

Abstract

3D reconstruction from images, the problem of reconstructing depth from images, is one of the most well-studied problems within computer vision. In part because it is academically interesting, but also because of the significant growth in the use of 3D models. This growth can be attributed to the development of augmented reality, 3D printing and indoor mapping.

Progressive stereo reconstruction is the sequential application of stereo reconstructions to reconstruct a scene. To achieve a reliable progressive stereo reconstruction a combination of best practice algorithms needs to be used. The purpose of this research is to determine the combination of best practice algorithms that lead to the most accurate and efficient progressive stereo reconstruction i.e the best practice combination.

In order to obtain a similarity reconstruction the intrinsic parameters of the camera need to be known. If they are not known they are determined by capturing ten images of a checkerboard with a known calibration pattern from different angles and using the moving plane algorithm. Thereafter in order to perform a near real-time reconstruction frames are acquired and reconstructed simultaneously. For the first pair of frames keypoints are detected and matched using a best practice keypoint detection and matching algorithm. The motion of the camera between the frames is then determined by decomposing the essential matrix which is determined from the fundamental matrix, which is determined using a best practice ego-motion estimation algorithm. Finally the keypoints are reconstructed using a best practice reconstruction algorithm. For sequential frames each frame is paired with the previous frame and keypoints are therefore only detected in the sequential frame. They are detected, matched and reconstructed in the same fashion as the first pair of frames, however to ensure that the reconstructed points are in the same scale as the points reconstructed from the first pair of frames the motion of the camera between the frames is estimated from 3D-2D correspondences using a best practice algorithm.

If the purpose of progressive reconstruction is for visualization the best practice combination algorithm for keypoint detection was found to be Speeded Up Robust Features (SURF) as it results in more reconstructed points than Scale-Invariant Feature Transform (SIFT). SIFT is however more computationally efficient and thus better suited if the number of reconstructed points does not matter, for example if the purpose of progressive reconstruction is for camera tracking. For all purposes the best practice combination algorithm for matching was found to be optical flow as it is the most efficient and for ego-motion estimation the best practice combination algorithm was found to be the 5-point algorithm as it is robust to points located on planes.

This research is significant as the effects of the key steps of progressive reconstruction and the choices made at each step on the accuracy and efficiency of the reconstruction as a whole have never been studied. As a result progressive stereo reconstruction can now be performed in near real-time on a mobile device without compromising the accuracy of reconstruction.

Keywords

Stereo Reconstruction, Progressive, Real-Time, Keypoint Detection, Keypoint Matching, Ego-Motion Estimation, Keypoint Reconstruction, Best Practice, Empirical Assessment.

I dedicate this thesis to my mom Pam. This thesis is the culmination of her selfless sacrifices over the past 24 years to ensure I get a good education. Thank you mom. Special mention must also go to my uncle Gary. Thank you for mentoring me in my entrepreneurial projects and for being patient and understanding of my academic commitments.

Acknowledgements

This thesis would not have been possible without the guidance of Dr Sithole. His ability to understand my questions and get me to think "out of the box" has been remarkable. Throughout the past two years Dr Sithole has introduced me to the most exciting field of work. He has believed in my abilities right from the beginning when he suggested I perform my research in a computer science related field, coming from a different background this was a challenging task however looking back on the skills I have acquired I will be forever grateful to him.

Contents

Chapter 1

Introduction

1.1 Introduction

Today 3D reconstruction from images is commonplace. The growth of augmented reality, 3D printing and indoor mapping has renewed interest in reconstruction algorithms, in particular algorithms which can be implemented on the mobile device.

3D reconstruction from images is a passive method of reconstruction i.e. depth is measured indirectly. Unlike active methods which measure depth by emitting radiation onto the scene using a laser scanner or by projecting a light onto the scene using a structured light scanner, passive methods do not emit any radiation themselves and instead measure ambient radiation using a digital camera and then apply techniques on the resulting images to determine depth.

Cameras can capture images within microseconds and are thus dramatically faster than active methods. Furthermore they are cheaper, smaller and lighter in weight and power and are easily incorporated into a mobile device. For these reasons passive methods are more popular. There are various techniques which can be used to measure depth from images such as shape-from-silhouette, depth from defocus and structure from motion (SFM) but the most common is stereo reconstruction.

Stereo reconstruction involves computing depth from a pair of overlapping images captured either by two cameras at the same time or by one camera at a different time instant. Firstly the pair of images are undistorted to remove lens distortion (undistortion), secondly keypoints are detected in each image (keypoint detection), thirdly keypoints are matched (keypoint matching), fourthly the correspondences are used to determine the translation and rotation of the camera between the images (ego-motion estimation) and lastly the correspondences are reconstructed to form a sparse point cloud of the scene

(keypoint reconstruction). A in depth background to stereo reconstruction and the best practice algorithms available for each key step can be read in appendix C.

A scene can be incrementally reconstructed in 3D by performing stereo reconstructions as images are added one by one, for example video frames. This is referred to as progressive stereo reconstruction. There is a great interest in performing progressive stereo reconstruction on a mobile device in real time, so that the computation can keep up with the pace at which new images are added. This is often done so that one can ensure that the structure of a scene is adequately captured, which is advantageous when compared to SFM as with SFM if the acquisition is not pleasing the whole reconstruction will need to be redone using an updated set of images. Furthermore often real time reconstruction is done using online methods which require lower resolution imagery and hence result in weaker reconstruction accuracies.

The accuracy and efficiency of stereo reconstruction depends on the accuracy of the camera poses and the intersection angle of camera rays (proportional to accuracy). The camera poses are determined from point correspondences and therefore their accuracy depends on the accuracy of keypoint detection, keypoint matching and ego-motion estimation. The intersection angle of camera rays is proportional to the baseline distance between frames and therefore only frames which have a sufficient baseline distance i.e keyframes are reconstructed (keyframe selection). The key steps of progressive stereo reconstruction can be seen below in figure 1.1.

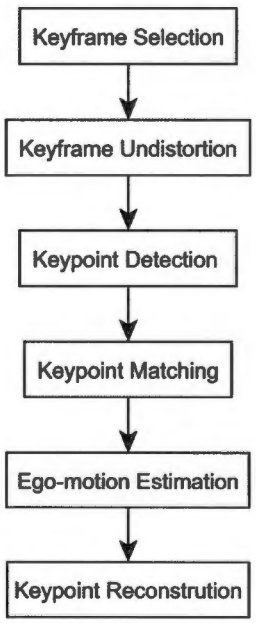


FIGURE 1.1: The key steps of progressive stereo reconstruction

The accuracy of progressive stereo reconstruction is not guaranteed due to a lack of global information (i.e the orientations and positions of the cameras at all frames are not known as the the frames are acquired progressively) and real time implementation requires the algorithm to be computationally efficient.

This research regards the term state-of-the-art as being representative of the latest algorithms (for example equivalent to nightly build in software development) and best practice to being the tried and tested more mature and robust algorithms.

The confidence in the accuracy and efficiency of progressive stereo reconstruction is improved when best practice algorithms are used. Furthermore best practice algorithms are advantageous as code to implement them is readily available in existing software libraries.

Therefore in order for a reliable progressive stereo reconstruction best practice algorithms and the selection of such algorithms for each key step that result in the most accurate and efficient reconstruction need to be used i.e the best practice combination.

1.2 Previous Work

Recently a large focus has been on improving the accuracy and efficiency of progressive stereo reconstruction by developing new keyframe selection and camera pose determination algorithms as will be demonstrated below.

Restricting reconstruction to keyframes improves firstly the accuracy of reconstruction as the distance between frames is proportional to the size of the intersection angle of camera rays and secondly the efficiency of reconstruction as redundancy is reduced.

Determining the camera poses more accurately directly improves the accuracy of reconstruction and as a result improves efficiency as computationally exhaustive optimization algorithms such as the bundle adjustment are not required to ensure accuracy.

A brief background to works presenting new keyframe selection and camera pose determination algorithms is given below. A detailed account of the works is provided in chapter 2.

1.2.1 Keyframe Selection Algorithms

Kang and Medioni (2015) made every tenth frame a keyframe. Ahmed et al. (2010) found keyframes that satisfied a minimum baseline length, contained point correspondences

that were not degenerate and ensured that the frames were not blurred due to video motion.Rashidi et al. (2013) extended on Ahmed et al. (2010) and presented a post capturing algorithm that determined the optimal number of keyframes required to ensure sufficient coverage of the scene. Mouragnon et al. (2006) determined keyframes as frames having over four hundred point correspondences between them. Pradeep et al. (2013) projected all scene points into each frame and if the amount of reprojected scene points fell below a threshold value the frame was labeled a keyframe. Zakharov and Barinov (2015) modeled the relationship between baseline length and number of correspondences and then determined the optimal baseline distance.

1.2.2 Camera Pose Determination Algorithms

Camera pose determination algorithms have focused on improving the accuracy of point correspondences. Douxchamps and Macq (2004) used perspective distortions to improve correspondences. Xiang and Sheng-yong (2011) manipulated the distance to and normal of each scene point in order to reduce the reprojection error and find more accurate correspondences. Geiger et al. (2011) used a stereo pair of cameras and matched correspondences between all four frames in a circle to ensure accurate correspondences. Se and Jasiobedzki (2006) matched features to a database of detected features to improve the accuracy of correspondences.

1.2.3 Other

Other notable research in the field of stereo reconstruction is as follows: Teng (2014) included measures of surface curvature to improve 3D reconstruction. Bao et al. (2014) used image segmentation to assist with the layout of a cluttered room. Brilakis et al. (2011) presented a novel approach to smooth reconstructed data. Pollefeys et al. (1999) included correspondences over a selection of views to determine the pose of the camera at a point accurately and Wang et al. (2012) performed stereo reconstruction on images captured from a stationary view point but under different lighting conditions.

As can be seen there has been considerable work on improving the accuracy and efficiency of reconstruction by developing new keyframe selection and camera pose determination algorithms. However there has been no work on measuring the effect different best practice algorithms have on the accuracy and efficiency of the reconstruction and therefore the best combination of best practice algorithms i.e the best practice combination is still not known.

1.3 Problem Statement

The accuracy and efficiency of progressive stereo reconstruction depends on the combination of best practice algorithms used. The combination that results in the most accurate and efficient progressive reconstruction is not known.

1.4 Objectives

To determine the combination of best practice algorithms that lead to the most accurate and efficient progressive stereo reconstruction i.e the best practice combination.

1.5 Research Questions

1. What is the common progressive stereo reconstruction algorithm?
2. What combination of best practice algorithms result in the most accurate and efficient reconstruction?

1.6 Methodology

A literature review will be performed on recent progressive stereo reconstruction algorithms to determine how others have improved the accuracy and efficiency of reconstruction

Once the literature review has been completed a stereo reconstruction algorithm that can reconstruct a scene progressively will be proposed and a selection of best practice algorithms to be tested will be identified for each key step .

The accuracy and efficiency of the reconstruction cannot be determined theoretically and therefore a sample dataset is used so that they can be tested by observation (empirical) and not hypothesis . The best practice combination will therefore be determined by doing an empirical study using different combinations of best practice algorithms.

1.7 Scope Of Research

The scope of this research is restricted to stereo reconstruction from images. Monocular images are used so that the reconstruction can be implemented on a mobile device and is not device restricted. The research assumes that the C++ libraries used are optimized.

1.8 Outcomes Of Research

This research will result in a report, the accuracies and efficiencies of various combinations of best practice algorithms i.e a benchmark and C++ code to perform progressive stereo reconstruction for a desired combination.

1.9 Structure Of Report

Chapter 2 is a literature review, a summary of significant works relating to the problem are discussed so that an understanding to the problem can be developed.

Chapter 3 presents the method which answers both research questions.

Chapter 4 presents the accuracy and efficiency of the reconstruction for varying combinations of best practice algorithms.

Chapter 5 presents the conclusion. A summary of the main results is presented along with their implications. Recommendations are given as well as suggestions for future research .

Appendix A presents a background to digital image processing and relevant best practice algorithms.

Appendix B presents a background to single view geometry and relevant best practice algorithms.

Appendix C presents a background to two-view geometry and relevant best practice algorithms.

Chapter 2

Literature Review

In the chapter below recent stereo reconstruction algorithms will be presented. There has been considerable work on progressive stereo reconstruction algorithms and a particular focus has been on improving the accuracy (as the accuracy of reconstruction is not guaranteed due to a lack of global information) and efficiency of reconstruction (so that reconstruction can take place in real time).

Recently a large focus has been on improving the accuracy and efficiency of progressive stereo reconstruction by developing new keyframe selection and camera pose determination algorithms. Therefore papers on progressive stereo reconstruction algorithms that implemented new keyframe selection algorithms or camera pose determining algorithms were reviewed. A thorough description of the reconstruction algorithm and the new keyframe selection/camera pose determining algorithm is given.

Therefore the works reviewed are divided into three sections namely works that deal with keyframe selection, works that improve the accuracy of camera poses and other works which include novel algorithms.

Where algorithms employed are best practice they are simply mentioned and detailed descriptions can be found in the appendices.

2.1 Keyframe Selection

Videos/live feeds are streams of up to sixty frames per second (FPS). The higher the frame rate the smaller the distance between each frame i.e the baseline distance. The advantages of shorter baselines are that the risk of viewpoint related occlusions is reduced and therefore more correspondences can be found and a denser reconstruction can be achieved. The disadvantage however is that the intersection angles of camera rays (i.e the accuracy of reconstruction) is smaller as it is proportional to the distance between frames. Therefore to improve the accuracy of reconstruction the distance between frames is increased by skipping frames. The frames remaining are known as keyframes. Only reconstructing keyframes not only improves the accuracy of reconstruction but also the efficiency as redundancy is reduced.

2.1.1 Progressive 3D Model Acquisition With A Commodity Hand-Held Camera

Kang and Medioni (2015) presented a novel progressive stereo reconstruction algorithm that detected and densely reconstructed keyframes of on object before refining the reconstruction.

For the first two frames Scale-Invariant Feature Transform (SIFT) features were detected, matched and reconstructed using the 5-point algorithm. Dense optical flow was then performed on the right frame and each frame thereafter to guide the matching of SIFT feature points in the next frame and to grow a track of correspondences. A local search was performed for three pixels around the optical flow predicted position and the closest SIFT point was the match if the dot product of their normalized descriptors was larger than 0.8. If there were multiple possible matching SIFT points the area had a homogeneous texture and no match was made.

At each frame the camera pose that resulted in the lowest reprojection error was determined from 3D-2D correspondences using the Levenberg-Marquardt algorithm. A frame became a keyframe when a track of correspondences spanned more than ten frames and covered a minimum distance from the last keyframe. Generally points are reconstructed when the angle of rays is greater than ten degrees however Kang and Medioni (2015) reconstructed points only at keyframes, furthermore at keyframes SIFT features that were detected but hadn't been matched with the previous frame begun their own track. A bundle adjustment was also performed at keyframes.

So far a 3D point cloud consisting of sparse SIFT feature points had been reconstructed and the camera pose at each frame had been determined. To fully reconstruct the

geometry of the scene another 3D point cloud was composed consisting of local planar patches which densely covered the surface. Planar patches were manually found and recorded by their center position and normal, each patch was thereafter tracked using optical flow and reconstructed at keyframes in similar fashion to SIFT points. A mesh was then built over the patched point cloud. The patched point cloud obtained was composed of sparse patches and therefore the mesh was smooth and structural details were lost.

To improve the capturing of structural details the depth map for each keyframe was refined by incorporating information from neighboring keyframes. Furthermore reconstructed patches were warped into four neighboring frames to check if their depth there was the same, if the difference in depth was larger than a set value for more than two neighboring frames the patch was filtered out as erroneous. Mesh models were generated through Poisson surface reconstruction.

2.1.2 An Algorithm For 3D Object Reconstruction From Video Using Stereo Correspondences

Zakharov and Barinov (2015) presented a novel progressive monocular stereo reconstruction algorithm that reconstructed a known object from video frames incorporating prior knowledge of the object.

The camera was calibrated and moved along a straight line sideways with its optical axis perpendicular to the direction of motion. Therefore the extrinsic and intrinsic parameters of the camera were known.

Scale-Invariant Feature Transform (SIFT) features were detected and feature based matching was performed. As the optical axis was perpendicular to the direction of motion the matched points were reconstructed using triangle geometry (simple stereo).

To improve the accuracy of reconstruction priori information about the object being reconstructed was used. All possible objects which could have been in the scene were given a vector of the following five parameters of prior knowledge: the number of primitives on the surface of the object, the type of primitive, the shape of the primitive and the texture of the object. The probability of the reconstruction being of each possible known object was determined using the Markov Chain Monte Carlo method.

Furthermore a comparison was done between the amount of correspondences detected and the baseline distance between frame/intersection angle of camera rays, with respect to the initial position of the camera. The analysis showed that the number of correspondences decreased as the distance between frames/intersection angle with respect to

the initial position of the camera increased. A baseline distance corresponding to an intersection angle of 25 degrees was then recommended. This can be seen in figure 2.1

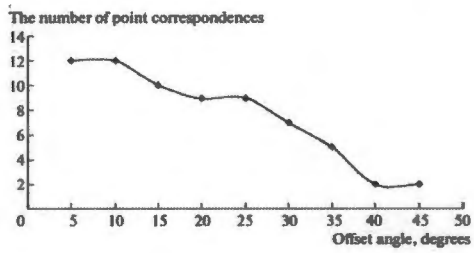


FIGURE 2.1: Graph of the number of detected correspondences vs baseline distance/ intersection angle of camera rays between frames. The relationship is inversely proportional Watson et al. (2012)

2.1.3 MonoFusion: Real-time 3D Reconstruction Of Small Scenes With A Single Web Camera

Pradeep et al. (2013) presented a novel progressive monocular stereo reconstruction algorithm for a simple calibrated consumer web camera .

In the beginning for each frame that is captured keypoints were detected using the FAST feature detector. Patches were extracted around each feature and intensity based matching was performed with the previous frame using the Zero Mean Normalized Cross Correlation Score. The affect of the zero mean allowed the correlation to perform well in patches that had large radiometric differences. RANSAC was used to refine the matches and the best five matches were used to compute the essential matrix using the 5-point algorithm. The essential matrix was then decomposed to determine the translation and rotation of the camera.

When the translation between frames i.e baseline length was above a predetermined distance the matched features were reconstructed. The reconstructed points were recorded together with the patch from their corresponding image point in the right frame.

For every frame thereafter two tasks were executed. Firstly the translation and rotation between the previous frame was estimated and used to reproject all the current reconstructed points into the new frame. Three points that reproject close to detected features were noted and the the camera pose that resulted in the lowest reprojection error was determined from 3D-2D correspondences using the Levenberg-Marquardt algorithm. If the amount of reprojected scene points fell below a threshold value the frame was labeled a keyframe.

Secondly in order to obtain a dense reconstruction stereo matching was performed between the current frame and the furthest keyframe which had over sixty five percent of the same reprojected scene points. This ensured the largest overlap possible and avoided an extremely short baseline. The stereo reconstructions were registered using a volumetric fusion approach.

2.1.4 Robust Key Frame Extraction For 3D Reconstruction From Video Streams

Ahmed et al. (2010) presented a novel progressive monocular stereo reconstruction algorithm for a video stream that implemented a novel approach to automatically select keyframes that are robust to motion blur and point degeneracies, as well as have a significant baseline length to ensure accurate reconstruction. By only reconstructing key frames the 3D reconstruction process was computationally more efficient. According to Ahmed et al. (2010) only seven to eleven percent of video frames are necessary to generate a high quality point cloud.

The first frame was set as a keyframe and the successive frame was determined by the following steps. The first step was to determine the frames whose correspondence ratio constraint was between an upper and lower threshold. The correspondence ratio constraint is the number of frame-to-frame correspondences over the total number of keypoints. It is inversely proportional to the camera motion and therefore is a good proxy for baseline length. It can also vary rapidly depending on the uniformity of the scene, scenes with a large texture change will have a rapid change in ratio.

The next step was refining these candidate frames by checking whether the point correspondences were degenerate. If point correspondences are degenerate then the the relationship between frames is better defined by a homography than a fundamental matrix. To assess this relationship the Geometric Robust Information Criterion (GRIC) was used. If the GRIC score for the homography model was lower than the GRIC score for the fundamental matrix the frame was eliminated as a candidate keyframe.

Each remaining candidate frame was then given a value comprised of the normalized GRIC difference (the difference in fundamental and homography GRIC scores divided by the homography GRIC score) and the point-to-epipolar line cost (PELC). High PELC values tend to occur in blurry images (motion blur is caused by jerky motions of the camera), therefore it was necessary to include it as an additional criterion for keyframe selection.

The algorithm was tested on a synthetic and real data set. On the synthetic data the method was able to identify the true degenerate cases perfectly with only 3 false positives among 93 frames tested. The real data was obtained using a calibrated video camera.

Keypoints were matched between each keyframe. For the first two frames the essential matrix was determined and decomposed to determine the cameras pose and the keypoints were reconstructed. For subsequent keyframes the cameras poses that resulted in the lowest reprojection errors were determined from 3D-2D correspondences using the Levenberg-Marquardt algorithm and the keypoints were then reconstructed.

The root mean reprojection error for each frame was computed and then the min,max,mean and standard deviation statistics over the entire sequence were computed. Lower mean reprojection errors and variance were noted in comparison to uniform sampling of keyframes.

2.1.5 Optimized Selection Of Key Frames For Monocular Videogrammetric Surveying Of Civil Infrastructure

Rashidi et al. (2013) presented a novel progressive monocular stereo reconstruction algorithm for a video stream that implemented a novel approach to automatically select keyframes.

Rashidi et al. (2013) extended on the work of Ahmed et al. (2010) by improving the removal of frames with motion blur and ensuring the scene was fully reconstructed by determining the minimum number of keyframes needed, so to avoid redundant reconstructions and improve the computational efficiency.

First off frames that were blurred were detected by using the BluM metric threshold. This threshold was measured by taking a sample of high quality un-blurred images and determining the average BluM metric value ,this was then the threshold and images with a metric value greater than the threshold were classified blurry and removed.

Secondly keyframes were found. The first frame became a a keyframe, subsequent frames with sufficient overlap and baseline length were determined using the correspondence ratio (number of frame-to-frame correspondences over the total number of keypoints). These frames became candidate keyframes. Candidate frames that lead to degenerate cases or large reprojection errors were removed by comparing the GRIC scores for homography and the fundamental matrix, as explained by Ahmed et al. (2010).

As the fundamental matrix and homography matrix were known RANSAC was then performed and the percentage of inlier's to the total number of correspondences was

determined for the fundamental and homography case. Each candidate frame was then scored, the score was calculated by determining the difference in inlier percentage for each case divided by the percentage of inlier's for the fundamental matrix and then multiplying it by the standard deviation calculated from measuring how uniform the distribution of features over the frame was. The final frame was the one which has the highest score. This frame then became the starting keyframe and the process continued.

The number of keyframes necessary to give a thorough reconstruction of the scene was determined before. The correspondence ratio that lead to this number of keyframes was determined experimentally. If the number of keyframes detected was out of the range then the ratio was readjusted and the algorithm was run again.

Finally the keyframes were reconstructed, feature matching was performed using Speeded Up Robust Features (SURF). The cameras pose at the second frame was determined using the 5-point algorithm, the cameras pose for frames thereafter was computed using the Direct Linear Transform (DLT) technique inside a RANSAC procedure and points observed by the new frames were reconstructed before sparse bundle adjustments was performed. Once all keyframes had been reconstructed the extrinsic and intrinsic parameters of all the cameras were entered into the Patch-based Multi-view Stereo Software and a dense point cloud was reconstructed.

2.1.6 Real Time Localization And 3D Reconstruction

Mouragnon et al. (2006) presented a 3D reconstruction algorithm that included a novel variant of the bundle adjustment which was more computationally efficient than the standard bundle adjustment yet just as accurate.

This was achieved by decreasing the number of parameters optimized. In order to decrease the amount of parameters a local bundle adjustment was performed each time a new set of scene points were reconstructed and thus a parameter dense post processing bundle adjustment was avoided.

Video frames were captured at a rate of 7.5 frames per second using a calibrated camera. Harris corners were detected in each frame and intensity based matching between each pair of frames was done using the Zero Normalized Cross Correlation score.

Three initial frames were chosen that were as far apart from each other as possible yet still had over four hundred matching points between the first and second frame and three hundred matching points between the first and third frame. This was in order to ensure a sufficiently large overlap so the epipolar geometry could be computed accurately. The camera was centered at the origin of the world coordinate system and the motion of

the camera through the first three frames was calculated using the 5-point algorithm and a RANSAC approach. The correspondences between the first and third frame were reconstructed.

At each frame thereafter the camera pose that resulted in the lowest reprojection error was determined from 3D-2D correspondences using the Levenberg-Marquardt algorithm.

Frames were either defined as normal frames or keyframes. A normal frame had more than four hundred matched points with the previous keyframe. If the next frame was a normal frame it was matched with the last keyframe and the motion of the camera between frames was computed. If the next frame was not a normal frame the frame before it was made a keyframe and the matched points between the two keyframes were reconstructed and a local bundle adjustment was performed - adding processing time.

The local bundle adjustment reduced the number of calculated parameters by optimizing only the extrinsic parameters of the last n cameras and taking account of 3D reprojections in the last N frames. This can be seen in figure 2.2.

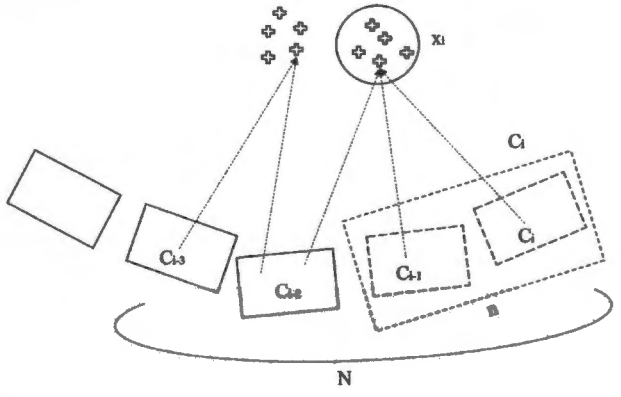


FIGURE 2.2: Local bundle adjustment when camera C_i is added. Only the surrounding n points and cameras are optimized, however the 3D point reprojections in the last N images are also used. In the figure the squares represent the image planes and the crosses represent the scene points.

By experimentation setting n = 3 and N = 10 the algorithm was 25 times more computationally efficient than the global bundle adjustment.

As the 3D reconstruction algorithm was an inside-looking-out approach the trajectory mapped was compared to GPS measurements taken (as satellites were present in the sky free from occlusions). The results showed that the algorithm was well suited to long scene reconstruction in terms of computing time, precision and robustness. The estimated mean 3D position error compared to global bundle adjustment was 0.29m.

2.2 Accuracy of Camera Poses/Point Correspondences

The accuracy of camera poses depends on the accuracy of point correspondences, as the camera poses are derived from the point correspondences. Determining the camera poses more accurately directly improves the accuracy of reconstruction and as a result improves efficiency as computationally exhaustive optimization algorithms such as the bundle adjustment are not required to ensure accuracy.

2.2.1 Integrating Perspective Distortions In Stereo Image Matching

Douxchamps and Macq (2004) presented a progressive stereo reconstruction algorithm which combined keypoint matching and reconstruction into one step by using perspective distortion.

Severe perspective distortion can weaken reconstruction and therefore most progressive reconstruction algorithms deal with it by applying feature based matching instead of intensity based matching or by simply removing it through rectification. The algorithm presented used perspective distortion as the main source of information.

Two calibrated cameras with known exterior parameters were required. The equation of a plane which was tangent to an object in the scene at a point was estimated. The intersection of the first camera ray with this plane gave an estimate of the point P. This can be seen in figure 2.3

To find the true value for point P the following steps were followed: a neighborhood of image points from the first image were projected onto the plane, using the extrinsic parameters of each camera they were then reprojected into the other images and the intensity values of the neighborhoods were then matched using the sum of absolute differences (SAD) correlation score. The plane parameters were then varied to find the parameters that gave the minimum SAD i.e the true parameters of the plane. The intersection of the first camera ray with the plane was the true value for point P. This method was computationally intensive.

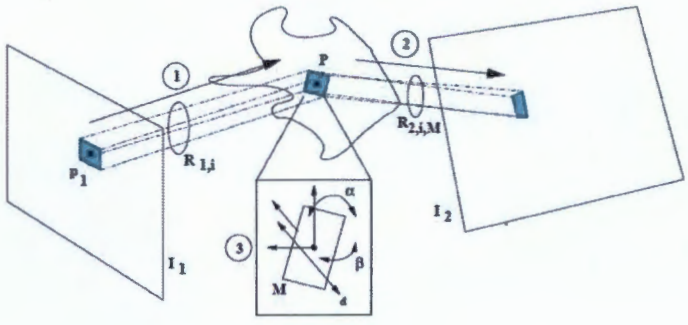


FIGURE 2.3: A neighborhood of image points is projected onto a tangent plane and the orientation of the plane is optimized so that the reprojection of the points onto surrounding images has the minimum sum of absolute differences. In the above figure the quadrilaterals represent the image planes and P represents the point being reconstructed. (Douxchamps and Macq, 2004).

2.2.2 A 3D Image Reconstruction Model From Multiple Images

Xiang and Sheng-yong (2011) presented a reconstruction refinement mechanism for getting better corresponding points and thus more accurate camera poses and a more accurate reconstruction.

A sequence of images were captured around an object and keypoints were reconstructed. The reconstructed points were reprojected into all images and correspondences were found with points in a reference image. The reprojection error was then calculated by measuring the distance between the reprojected points and their epipolar lines.

The feature points in the reference image were then sub-sampled over a grid to ten to twenty percent of their original number. To improve the accuracy of the sampled points their normal angle and their distance from the camera was altered. For each alteration the reconstructed point was reprojected into the images to measure the new reprojection errors. This can be seen in figure 2.4.

An image pyramid was formed. If the distance between the reprojected points at different levels of the image pyramid was larger than the reprojection error then the point after refinement was not good enough and was removed.

For each feature point in the reference image the corresponding point in the images with the largest normalized cross correlation score was selected as the matching point. Finally the refined set of corresponding points were used in a Sparse Bundle Adjustment (SBA) to calculate more accurate camera parameters and thus a more accurate reconstruction.

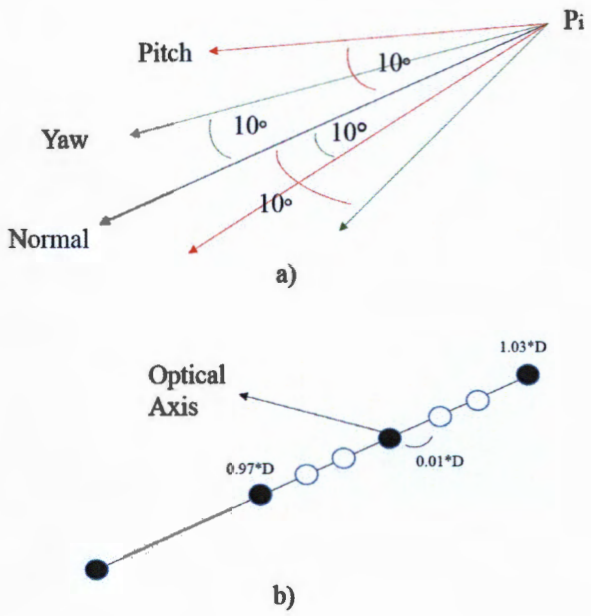


FIGURE 2.4: Manipulating the angle of the normal and reconstructed point depth in figures a and b respectively. In figure a P_i is the camera . In figure b the black and white circles represent possible positions of the camera as it is moved closer to and further away from the scene being captured.

2.2.3 StereoScan: Dense 3D Reconstruction In Real-Time

Geiger et al. (2011) presented a novel real-time progressive stereo reconstruction algorithm for high-resolution stereo sequences that focuses on achieving accurate point correspondences.

A smooth trajectory of a stereo pair of cameras was assumed therefore rotationally and scale invariant feature points such as Scale-Invariant Feature Transform (SIFT) or Speeded Up Robust Features (SURF) were not necessary. Instead keypoints were found by filtering the input images by blob and corner masks and applying non-max and non-min suppression to the filtered images. The resulting keypoints either belonged to one of four classes (blob min,blob max,corner max,corner min). To improve matching horizontal and vertical Sobel filters were applied to windows around these key points.

To perform matching accurately the key points in consecutive stereo pairs were matched in a circular manner. Each pair included a left and right frame. The circle begun by matching keypoints in the current left frame with keypoints in the previous left frame, then points in the previous left frame were matched with points in the previous right frame, which were matched with points in the current right frame, which were matched with points in the current left image again. The circle match got accepted if the last

keypoint coincided with the first keypoint, therefore the match was an accurate one and could be used to estimate camera motion in the next step. Intensity based matching was performed using the sum of absolute differences (SAD) correlation measure.

To estimate the camera motion the keypoints were matched over two stereo pairs i.e four frames. The keypoints in the first stereo pair were reconstructed. Each reconstructed point was then reprojected into the left and right frames of the second stereo pair. The reprojection error was measured as the sum of the distances from the corresponding match point in the respective frames to the reprojected point. A RANSAC approach was followed and the camera pose which resulted in the smallest sum of reprojection errors was found after 50 estimations using 3 randomly drawn correspondences.

Dense stereo matching was then performed and disparity maps were computed. All the pixels from the disparity images were transformed into a common coordinate system according to the estimated camera motion.

To reduce the large amount of redundant information the reconstructed points of a previous stereo pair were reprojected into the current stereo pair frames. In the case a point fell onto a valid disparity both reconstructed points were fused together by computing their mean. This reduced storage and lead to improved accuracy as measurement noise was averaged out over several frames.

2.2.4 Photo-Realistic 3D Model Reconstruction

Se and Jasiobedzki (2006) presented a novel progressive stereo reconstruction algorithm for an off-the-shelf hand held stereo camera that focused on achieving accurate point correspondences. The method is currently being used by hardware manufacturing company MDA in their stereo camera ISM. The ISM camera contains a pair of calibrated cameras setup in stereo with a baseline distance of 12cm.

For each stereo pair dense stereo matching was computed. The coordinate system of the first stereo pair was chosen as reference coordinate system. Each stereo pair computed thereafter was transformed into this system. To determine the transformation the motion of the camera was required. To determine the motion SIFT features for each stereo pair were detected and matched. Because the stereo camera geometry was known a set of constraints such as the epipolar and disparity constraint were available which allowed the SIFT features for each stereo pair to be matched easily. The reconstructed coordinates of the matched SIFT features were computed and stored in a database.

The motion of the camera was recovered by matching these SIFT features with a database instead of from frame to frame, the camera movement that would bring each

SIFT feature into the best alignment with its matching feature was found by least squares. SIFT features that were detected and are not in the database were added to the database so that they may be used to recover camera motion for the next stereo pairs. This Simultaneous Localization and Mapping (SLAM) approach led to a 27.7 percent reduction in rover navigation error compared to each pair of frames being considered separately.

Once the reconstruction was complete it was meshed and color images from the stereo camera were used for texture mapping. The images which covered the most triangles were used to solve the problem of each triangle being observed by multiple images, this led to lower storage requirements and faster model loading speeds.

Se and Jasiobedzki (2006) also presented a auto-referencing algorithm for when a continuous scan is not possible but separate scans are. The SIFT database map for each model was used to estimate the parameters needed to align the models. A list of tentative matches were found and a random three tentative matches were chosen to compute the 6 degrees of freedom aligning parameters. RANSAC was used and the random selection was repeated until the three matches which proposed the alignment with the most support was found.

In indoor man-made environments there is often a lack of texture for dense stereo matching. Se and Jasiobedzki (2006) presented a solution to this by projecting a random dot pattern on the scene every 10 frames. The SIFT features determined during such a projection could not be used to determine the camera motion as the same pattern was projected each time, therefore the camera motion for this frame was interpolated from the motion at the adjacent normal frames.

2.3 Other

Below are a selection of works that present innovative approaches to image based reconstruction.

2.3.1 Hand-Held Acquisition Of 3D Models With A Video Camera

Pollefeys et al. (1999) presented a novel progressive stereo reconstruction algorithm that combined a number of state of the art algorithms so that the reconstruction could be performed using a uncalibrated camera with no restrictions on its motion.

Corners were detected in the frames using the Harris corner detector and intensity based matching was performed on patches around the corners by Normalized Cross Correlation. Once matching was performed RANSAC was used to refine the matches before the fundamental matrix could be determined. For the first frame the camera pose was set to zero and for the second frame the camera pose was calculated from the fundamental matrix. The point correspondences were then reconstructed using the optimum triangulation algorithm.

As the calibration matrix was unknown the points were reconstructed within a projective transformation of the true construction i.e the only aspect of the reconstruction that was preserved and therefore was true was collinearity. For each frame thereafter feature points were matched and matches corresponding to already reconstructed points were found resulting in a set of 3D-2D correspondences. The camera pose that resulted in the lowest reprojection error was determined from the 3D-2D correspondences using the Levenberg-Marquardt algorithm.

The determined camera pose was then used to see which existing scene points reproject onto the image, therefore finding additional matches to refine the estimation of the camera matrix. However this estimation of the cameras pose was not optimal, instead of just relating the frame with the previous frame Pollefeys et al. (1999) included a method that ensured that if a point got out of sight there was still a chance it could be reconstructed. The common solution to this was to match features with not only the previous frame but all frames. This is computationally exhaustive and Pollefeys et al. (1999) instead matched features with all close views. 3D-2D correspondences were generated for each close view and combined allowing a more accurate camera pose to be determined.

In order to upgrade the reconstruction from projective to similarity/metric the calibration matrix was determined by self calibration. Self calibration involves determining the

image of the absolute conic. It is advantageous as it allows the intrinsic parameters to vary during acquisition which is very useful in cases where the camera is equipped with zoom.

To achieve a dense reconstruction Watson et al. (2012) first rectified the images and then applied a standard stereo matching algorithm to determine depth maps. For some motions such as when the camera translates into a scene the epipole's are located in the image and standard rectification will not work so Pollefeys et al. (1999) devised a new approach to rectification that works for all motion, it involved using polar coordinates with the epipole at the origin.

The depth maps were interpolated using a parametric surface model, the smoothed surface was then approximated by a triangular wire frame mesh and its texture was mapped. The only thing Pollefeys et al. (1999) did not cover in there research was fusing the reconstructions together to improve the accuracy and eliminate artifacts at the boundaries.

2.3.2 Improving Three-Dimensional Point Reconstruction From Image Correspondences Using Surface Curvatures

Teng (2014) presented a novel stereo reconstruction algorithm that incorporated surface characteristics, specifically the Gaussian and mean curvature, to improve the accuracy of reconstruction.

Two cameras were setup with a 3.6 degree difference in orientation and 2 units apart. The surface was a further 60-85 units apart. At this configuration the rays of corresponding points are near parallel. Therefore slight errors in point correspondences often give rise to large reconstruction errors. Visually it is therefore an ideal configuration to see the improvements of using the surface characteristics.

The reconstruction was modeled as an optimization problem, the image correspondences produced data constraints and the Gaussian and mean curvatures made up the soft constraints. The soft constraints differed for each surface. For planes, cylinders and spheres the constraint equations were known. For unknown surfaces it was assumed that the surface was smooth and a general constraint was used. To determine the Gaussian and mean curvatures so the constraints can be computed for each point a paraboloid can be fit and curvature values determined off its surface, however to fit a paraboloid the normal of the surface is required and therefore a mesh is needed, to avoid creating a mesh quadratic fitting can be used, however this method often does not perform well in 3D reconstruction. Therefore Teng (2014) combined paraboloid and quadratic fitting.

Quadratic fitting was used to determine the normal of a point and then a paraboloid was fit to determine the Gaussian and mean curvatures and formulate the soft constraint.

The reconstruction algorithm worked as follows: corners were detected and matched with the help of optical flow and the cameras poses were determined. The image was segmented into surfaces so that the corresponding points which belonged to the same surface were known. The segmented regions were assumed to be smooth and the unknown surface curvature constraint was applied to reconstruct the 3D points. This method was computationally intensive and inappropriate for real time.

2.3.3 Generation Of 3D Sparse Feature Models Using Multiple Stereo Views

Watson et al. (2012) presented a novel progressive stereo reconstruction algorithm to assist a augmented reality vision system recognize an object in the world and overlay information.

Two calibrated cameras were setup in stereo 68mm apart. Therefore the extrinsic and intrinsic parameters of the cameras were known.

Correspondences were found by feature based matching using Speeded Up Robust Features (SURF) features. SURF features are very distinctive and therefore have a high dimensional descriptor vector. Nearest neighbor matching cannot deal with high dimensional data and approximate methods such as The Fast Library for Approximate Nearest Neighbour Matching (FLANN) are quicker but less accurate. Therefore a modified linear search was used as accuracy was more important than efficiency.

For each stereo pair the correspondences were reconstructed by simple triangle geometry (simple stereo). The Iterative Closest Point algorithm was then used to align the reconstructed points for each stereo pair. The resulting scene was denser and had a higher resolution in comparison to its wide baseline counterparts.

2.3.4 Progressive 3D Reconstruction Of Infrastructure With Videogrammetry

Brilakis et al. (2011) presented a novel progressive stereo reconstruction algorithm to reconstruct the geometry of an industrial site from a stereo camera stream so that object segmentation and classification could be performed to generate useful information which is necessary to solve complex problems i.e as-built modeling.

Speeded Up Robust Features (SURF) features were matched and reconstructed in each pair of video frames. A sparse point cloud was therefore formed for each pair of stereo frames. The sparse point clouds were merged by tracking the SURF feature points from one pair of frames into the other. The reconstructed coordinates of the tracked points were known allowing the rotation and translation of the camera to be determined and the point clouds aligned.

In order to thicken the sparse points clouds the fundamental matrix was calculated using the 8-point algorithm and new points were found and matched along the epipolar lines. To match new points an adaptive matching algorithm with the following properties was implemented: 1. The window size of the point in the first image was inversely propositional to the gradient (therefore larger windows were used in relatively uniform areas) 2. If the window was small (non-uniform area) its matching score was defined as the sum of the absolute differences of the red, green and blue colors. 3. If the window was large (uniform area) then the average depth of neighboring points was determined and used to assist finding matches.

Brilakis et al. (2011) furthermore presented a novel algorithm to smooth the reconstructed points i.e handle outliers, firstly in each image random points were selected and a small window around each point was checked for edge points, if no edge points were present the window represented a uniform area, the distance from each reconstructed point to its neighbors was then calculated and the point was moved so to reduce this distance i.e achieve co-planarity and smooth the data.

2.3.5 3D Model Reconstruction Algorithm And Implementation Based On The Mobile Device

Wang et al. (2012) presented a novel progressive stereo reconstruction algorithm that instead of using frames from multiple views used frames from a single view with varying lighting conditions.

The method was implemented on a android mobile device in a dark environment with light provided by the device's screen.

The normal value of each scene point was determined using the relationship between the known direction of light sources and the resulting intensities of images. The depth value for each scene point was calculated using polydrons of the normal values.

2.3.6 Understanding The 3D Layout Of A Cluttered Room From Multiple Images

Bao et al. (2014) presented a novel algorithm that used frames to estimate the 3D layout (e.g. floor, walls, and ceiling) of an indoor environment as well as identify the objects within it. Using images to understand the layout of a cluttered room is a great challenge in computer vision research. A room may be occupied by objects that are not present in a training set or/and the room walls may be occluded and unable to be observed directly.

Previous research has focused firstly on 3D reconstructing an unknown environment and then if desired recognizing objects after. The algorithm presented by Bao et al. (2014) moved away from separating the 3D reconstruction and object recognition and instead jointly estimated the 3D room layout using geometric and semantic cues which play complimentary roles in accurately recovering the geometry of the scene. For example when a room is very cluttered there will usually exist a large set of characteristic feature points which can yield a point cloud with reasonable density (geometric cue). Reconstructed points can help reason about the extent of the room. On the other hand when a room is clean and has little reconstructed points the image line segments and region segmentation results (semantic clue) can be used to obtain a good estimation of the rooms walls.

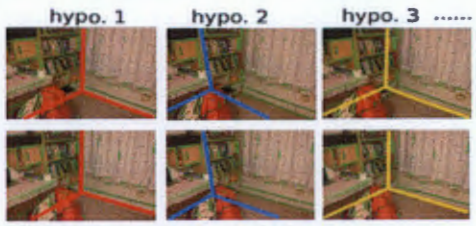
In order to estimate the 3D layout of an indoor environment as well as identify the objects within it a series of hypothesized room layouts and object configurations had to be tested. In order to determine the accuracy of each hypothesis a cost function was used. The cost function evaluated the likelihood of the room layout given the reconstructed points, estimated camera poses and region segmentations in every image.

First N images of the room were captured using a calibrated camera. Feature points were detected and matched and reconstructed. Region segmentation was performed on each image. The set of segmented regions were matched across images according to color, shape and appearance. Each segmented regions appearance was described by an appearance vector concatenating multiple cues. The appearance vector and a pre trained region classifier were used to determine the confidence that the region belonged to a class label (e.g. floors, walls, ceiling or objects) and region class confidence maps were formed.

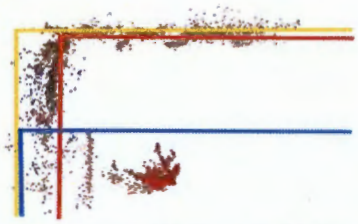
The Manhattan world assumption that the walls of a room must be perpendicular to one of three mutually perpendicular directions(dominant directions) was adopted. Dominant directions from the line segments were found (line segments are the boundaries of regions). Room corners may be occluded or corner detectors may have a weak response

so the estimated dominant directions were used to determine corners in the images where they intersect. The 3D locations of room corners was found by triangulation. Only a few of these were actually true corners. From the corner data set random samples were taken and a set of room layout hypotheses was generated, the size of the set was limited to 300 hypotheses. The room layout was assumed to be a cuboid so a layout hypothesis could be uniquely proposed using a number of corners. The k-means algorithm was used to cluster similar room layouts and keep only significantly different room layouts. The configurations were compared with the reconstruct points to see which hypothesis was compatible, this can be seen in figure 2.5.

(a) Input images and detected line segments



(b) Three wall hypotheses among many proposals.



(c) SFM points and hypotheses in 3D (top view)

FIGURE 2.5: Finding the room layout hypothesis that corresponds with the reconstructed points (Bao et al., 2014).

Once the correct room hypothesis was found the object hypothesis was generated by firstly following on from the room hypothesis and assigning labels to the reconstructed points – points close to the walls were labeled as non-objects and points that did not belong to walls were labeled as objects. The point labels were then reprojected onto the segmented regions. There may have been missing or wrong region labels as regions may have not carried sufficient reconstructed points and points labels may have been noisy.

Based on the labels initialized from reconstructed points a complete region classification was obtained. The missing labels were inferred and the wrong labels were corrected by enforcing appearance consistency. The regions labeled as objects were reprojected into 3D space if they had enough reconstructed points. The process of generating an object hypothesis can be seen in figure 2.6 .

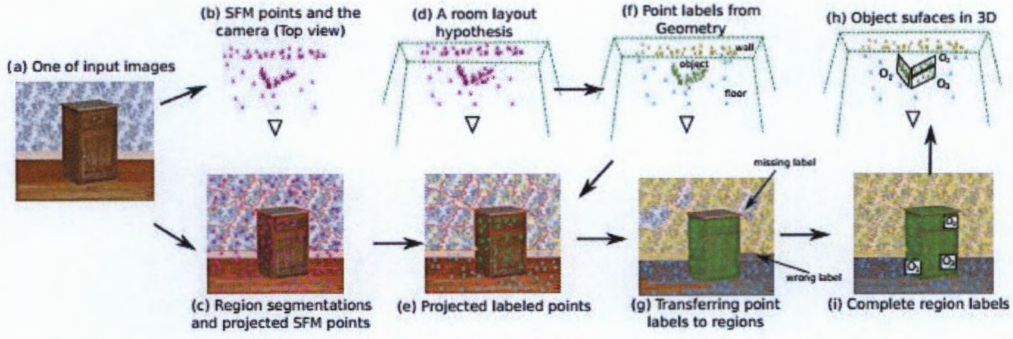


FIGURE 2.6: Corresponding Object hypothesis (Bao et al., 2014).

The accuracy of the room layout and object estimation was determined by projecting the estimations into each image, labeling every pixel and comparing towards the ground truth.

Chapter 3

Method

3.1 Introduction

This chapter begins by presenting the combinations of best practice algorithms tested. They will only receive mention as detailed explanations including equations are provided in the appendices. Thereafter a novel progressive stereo reconstruction algorithm is presented and explained thoroughly. A basic understanding of 3D reconstruction is assumed, where algorithms used are simply mentioned a detailed description of them is provided in the appendices. Finally the quantitative measures used to access the accuracy and efficiency of the reconstruction algorithm are presented.

3.2 Algorithms Tested

The following best practice algorithms were tested for each key step of progressive stereo reconstruction:

1. Keyframe selection: Minimum distance
2. Keypoint detection: Scale-Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF).
3. Keypoint matching: brute force feature-based matching, approximate feature-based matching and position (optical flow) matching.
4. Ego-motion estimation: for initial pair of frames: 5, 7 and 8 point fundamental matrix, for sequential frames: 3D-2D correspondences (iterative PnP algorithm)
5. Keypoint reconstruction: optimum triangulation

Therefore there were eighteen possible combinations of best practice algorithms which were tested by empirical analysis. Since the above algorithms are best practice and known to those experienced in the field they are simply mentioned throughout the report. Detailed descriptions can be found in appendix C.

As can be seen for the key steps "keyframe selection", "ego-motion estimation (for sequential frames)" and "keypoint reconstruction" there is only one best practice algorithm tested. "Keyframe selection" simply determines keyframes as frames which are separated by a "minimum distance". "Ego-motion estimation (for sequential frames)" simply determines the ego-motion (rotation and translation) between frames which results in the lowest reprojection error of reconstructed points into the right frame and "keypoint reconstruction" uses the optimum triangulation algorithm which ensures image points intersect by enforcing the epipolar constraint.

3.3 The Progressive Stereo Reconstruction Algorithm

The common progressive stereo reconstruction algorithm is broken into the following processes:

1. Camera calibration
2. Frame acquisition
3. Progressive stereo reconstruction (a novel drift robust approach is presented).

Frame acquisition and reconstruction are executed in independent threads allowing them to run concurrently.

A high level overview of the algorithm can be seen below in figure 3.1.

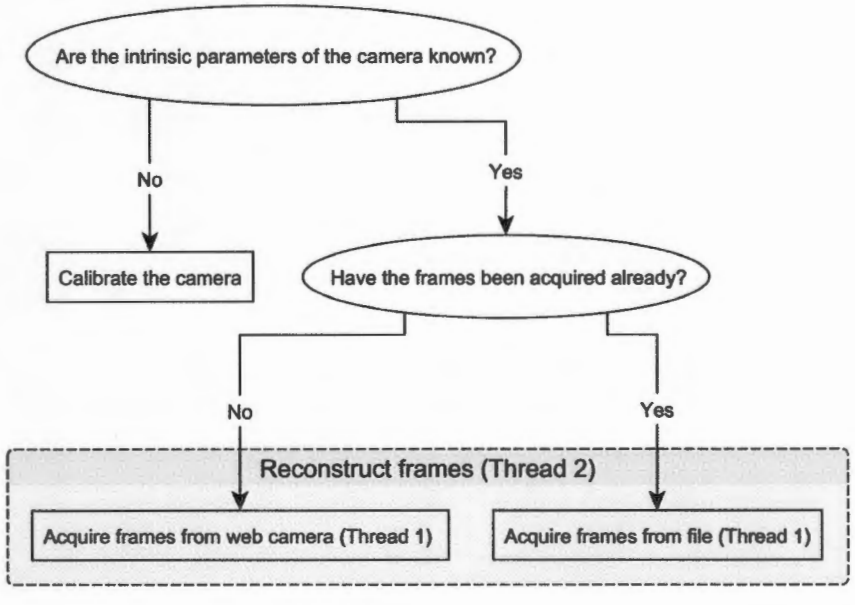


FIGURE 3.1: High level overview of program.

3.3.1 Camera Calibration

The calibration algorithm calibrates the camera i.e determines the intrinsic parameters by using the best practice moving plane algorithm. The moving plane chosen is a checkerboard. A description of the moving plane algorithm is provided in appendix B. Figure 3.2 below shows the corners of a checkerboard being detected using the algorithm.

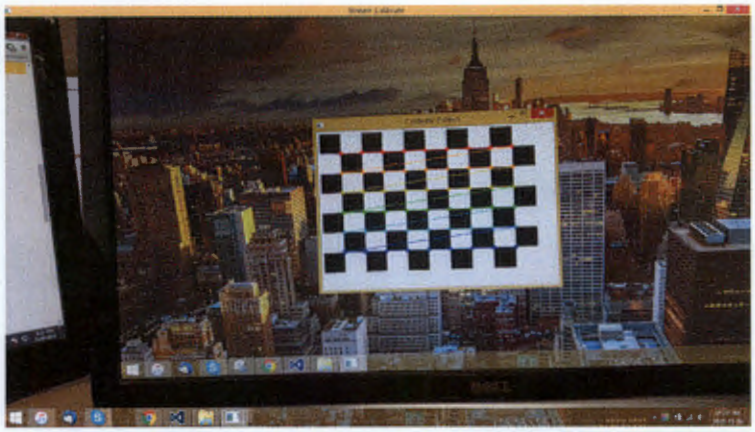


FIGURE 3.2: The above figure shows checkerboard corners detected using the moving plane algorithm. The checkerboard was displayed on an adjacent monitor and captured on a mobile device using the IP Webcam mobile application which enabled a real time stream.

The intrinsic parameters need to be known in order to obtain a euclidean reconstruction i.e. a reconstruction within a euclidean transformation (rotation and translation) of the real world. The cameras intrinsic parameters of the dataset used were provided and therefore camera calibration was not performed.

3.3.2 Frame Acquisition

The frame acquisition algorithm acquires frames from either a file location or from a web-camera/network camera(for example a cellphone with IP Webcam application). If the frames were acquired from a web camera they were grabbed every ten seconds to avoid small baselines and improve efficiency. If the frames were acquired from file small baselines were not a concern, as it was advised to avoid including such files in the file location. Therefore all frames acquired were keyframes and in essence the frame acquisition algorithm implemented the keyframe selection step. As a dataset was provided the frames were acquired from the file location.

3.3.3 Progressive Stereo Reconstruction

The frames were reconstructed using the below novel algorithm. An overview of the algorithm can be seen in figure 3.11. The algorithm differs for the first pair of frames and the frames thereafter.

First pair of frames:

1. The extrinsic parameters of the camera at the left frame were set to their true values so that an absolute reconstruction could be achieved.

2. The frames were undistorted.

   The effects of radial and tangential lens distortion were removed by applying an image transformation to the frames, in particular an inverse warping as described in appendix A. The warping transformation is a matrix of distortion coefficients .

3. Keypoints were detected and matching was performed.

   Keypoints were detected and matched using a best practice algorithm. For example SURF keypoints detected and matched using optical flow in the first two frames of the Strecha et al. (2008) *Fountain-P11* dataset can be seen below in figure 3.3 and 3.4 respectively.

FIGURE 3.3: Keypoint Detection: SURF Keypoints detected in the initial left and right frames

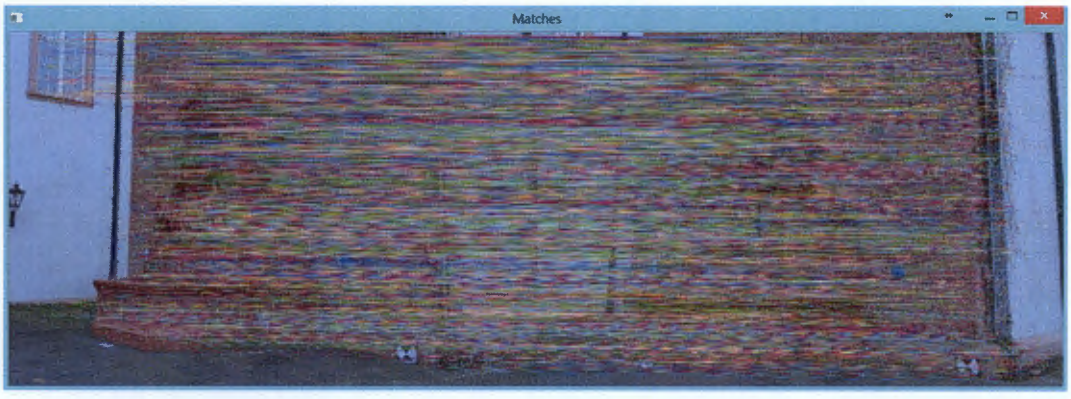


FIGURE 3.4: Keypoint Matching: SURF Keypoints matched between the initial left and right frames. The matched keypoints are joined by lines which are horizontal indicating side-ways motion of the camera.

4. The ego-motion i.e rotation and translation of the camera between frames was estimated.

   The ego-motion was estimated by decomposing the essential matrix as described in appendix C. The translation vector determined from the decomposition is normalized and thus the translation is determined up to scale. In order to determine the accuracy of sequential reconstructions the translation was scaled to the true distance between the frames so that the reconstructions were in the same scale as the ground truth data and an absolute reconstruction could be achieved.

   The essential matrix was determined from the fundamental matrix as the calibration matrix (intrinsic parameters) was known. The fundamental matrix was determined using a best practice algorithm. To ensure an accurate fundamental matrix a RANSAC approach was adopted. Random selections of points were chosen and the fundamental matrix was computed for each, furthermore the points that fell within a certain distance from their epipolar line were counted i.e inliers. The selection of points which had the highest count was used to compute the fundamental matrix again and were preserved.

   For example the SURF matches seen in figure 3.4 can be seen refined in figure 3.5. Furthermore the epipolar lines can be seen in 3.6.

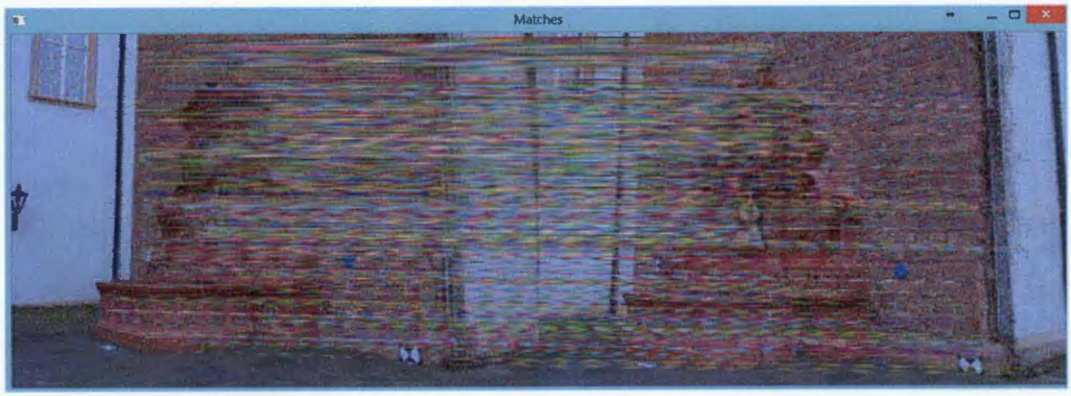


FIGURE 3.5: Refining Matches: Matches between the initial left and right frames refined using RANSAC.

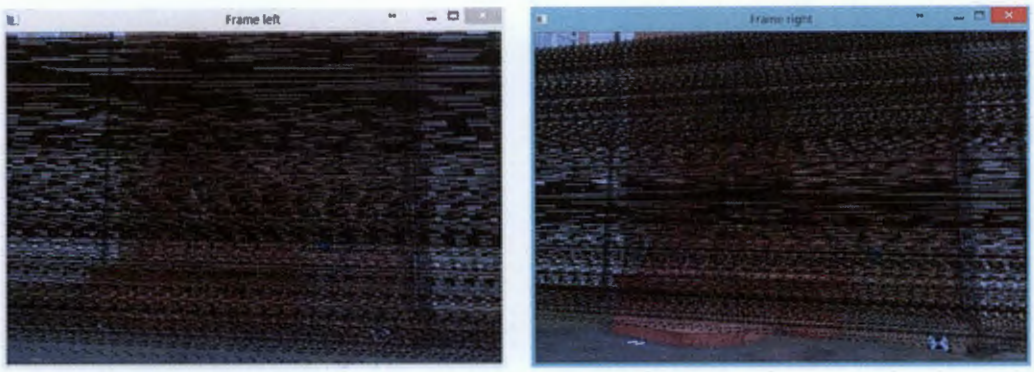


FIGURE 3.6: Epipolar lines in the initial left and right frames

5. The normalized camera matrix at the right frame was determined.

   The extrinsic parameters of the camera at the first frame were combined with the rotation and translation values of the camera between the frames to determine the normalized camera matrix at the second frame.

6. The matches were refined to avoid parallel rays (novel).

   As the frames were captured moving across the scene and not forward only matches in the center of the overlapping regions were reconstructed. This was a novel approach to avoid weaker camera intersection angles (and thus reconstruction accuracies) which occur at matches on the outer sides of the frames. To illustrate this two cameras were setup two meters apart with parallel optical rays as seen in figure 3.7 and a heat map of the intersection angles in the overlapping region was created as seen in figure 3.8.

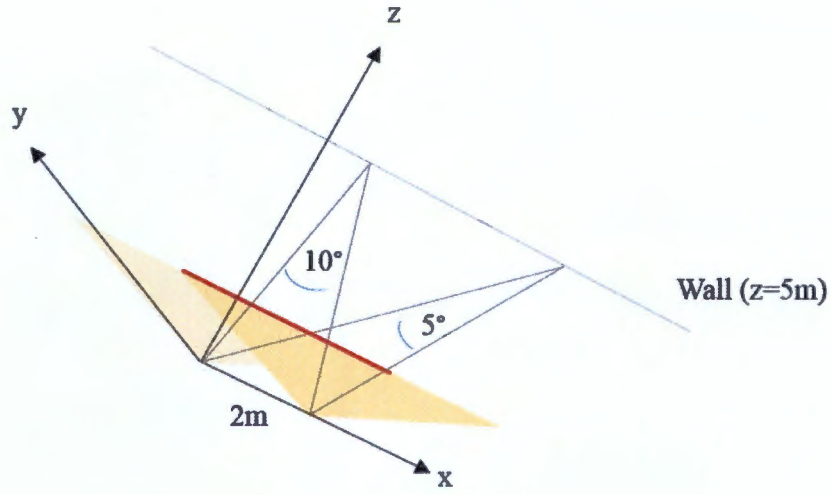


FIGURE 3.7: Two cameras were setup two meters apart with parallel optical rays facing a wall 5m in front. The red line represents the overlapping region.

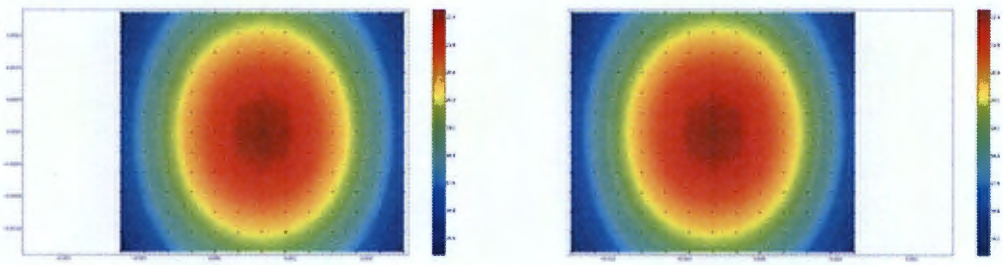


FIGURE 3.8: A heat map was created in Python using Matplotlib of the intersection angles for the left and right stereo frame. The intersection angle of camera rays increases towards the center of the overlapping region as seen in figure 3.7 as a red line.

7. The refined matches were reconstructed.

   The refined matches were reconstructed using the optimal triangulation algorithm as described in appendix C. Alongside each of the reconstructed points coordinates a map with a key equal to the number of the right frame used and a value equal to the position of the keypoint in the right frame was recorded.

Subsequent frames:

Each subsequent frame was paired with the previous frame i.e the previous frame was set as the left frame and the subsequent frame the right frame. Keypoints were therefore only detected in the right frame as the previous frames keypoints were set as the left frames keypoints.

The same initial four steps were implemented thereafter:

1. The normalized camera matrix at the right frame was determined.

   To ensure the progressive reconstruction of points was consistent in scale the extrinsic parameters of the camera were found using 3D-2D point correspondences. The extrinsic parameters i.e the orientation and position of the camera were found such that they would lead to the smallest reprojection errors. This was therefore an iterative minimization problem and the Levenberg-Marquardt algorithm was used. This is described in appendix C. Provisional values for the Levenberg-Marquardt algorithm were found by combining the rotation and translation of the camera between frames estimated in the previous step with the extrinsic parameters of the camera at the left frame.

   In order to determine the 3D-2D correspondences the following algorithm was used: for each refined match the map alongside each reconstructed point was searched to find the map whose key was equal to the position of the left matching frame and whose value was equal to the keypoint position of the refined match in the left frame . This 3D point and 2D point formed one correspondence. A minimum of four correspondences was needed.

2. The matches were refined to avoid parallel rays.

   The same step was followed as for the first pair of frames.

3. The refined matches were reconstructed (novel).

   As the extrinsic parameters of the camera were determined from 3D-2D correspondences their accuracy was no longer solely dependent on the accuracy of matching and also depended on the accuracy of the existing reconstructed points (which in turn were dependent on the accuracy of their corresponding camera poses and the intersection angle of cameras rays).

   Therefore to avoid a drift in errors duplicate matches were not reconstructed, instead for each already reconstructed match the map alongside their existing coordinates was appended with the right frame position of the new match as key and the matched keypoint position in the right frame as value. An example of the

3D-2D correspondences used to determine the extrinsic parameters of the cameras and the maps at each reconstructed point can be seen in figure 3.10.

The matches which had not be reconstructed before were then reconstructed in the same fashion as the first pair of frames. For example two views of the reconstructed points for the combination of best practice algorithms *SURF - OpticalFlow - 5-point* colored by reprojection errors can be seen below in figure 3.9 .

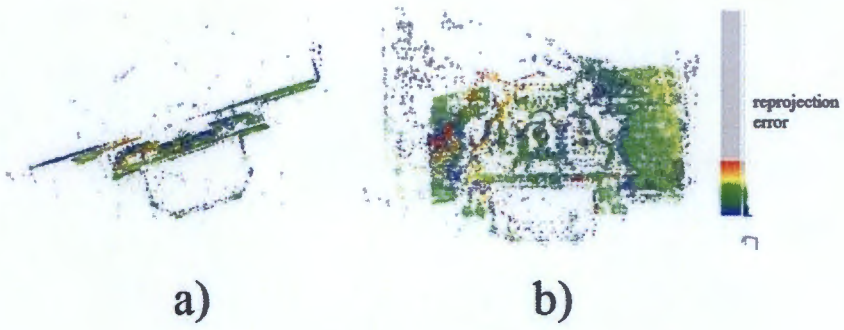


FIGURE 3.9: Two views, figures a and b, of the reconstructed points for the combination of best practice algorithms *SURF - OpticalFlow - 5-point* coloured by reprojection error

This was a novel approach and by avoiding a drift in errors computationally exhaustive optimization techniques such as the bundle adjustment were not required to ensure an accurate reconstruction. Therefore the efficiency of the algorithm was improved.

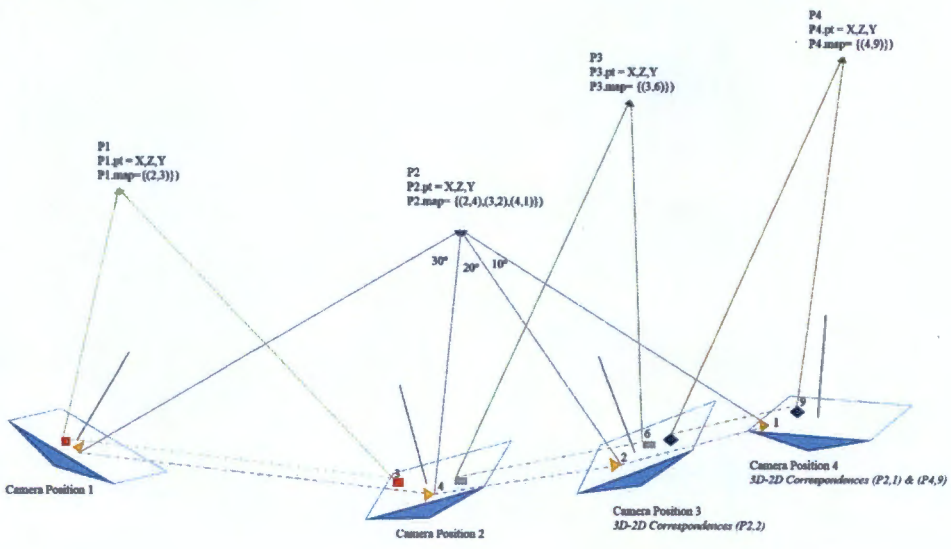


FIGURE 3.10: An overview of the 3D-2D correspondences used to determine the extrinsic parameters of the cameras at each frame and the maps formed at each reconstructed point.

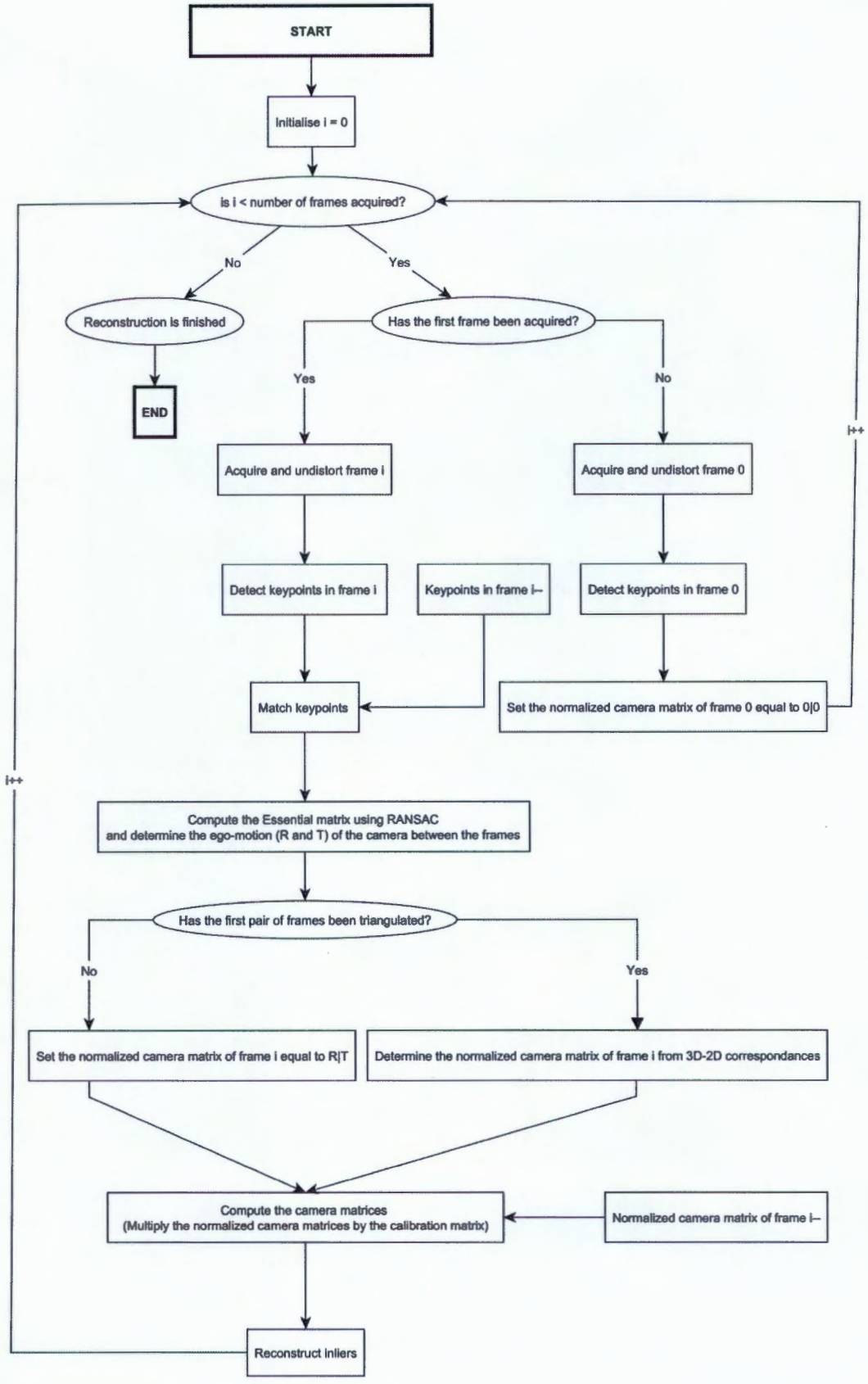


FIGURE 3.11: Flow chart of the reconstruction algorithm.

3.4 Measures Quantifying The Accuracy And Efficiency Of Reconstruction

For each combination of best practice algorithms the accuracy and efficiency of the progressive stereo reconstruction algorithm was quantified separately.

3.4.1 Accuracy

The accuracy of reconstruction was quantified by the reprojection error, the distance between the measured and true camera positions and the distance between the true and reconstructed points. The measures of accuracy can be seen in figure 3.12. As the reprojection error is measured in image pixels the later allows the accuracy in the world to be determined i.e in three dimensions.

The reprojection errors were computed by projecting each reconstructed point back into its corresponding image and measuring the image distance (pixels) between the reprojected point and the keypoint.

$$\sum_i d(p_i,\hat{p}_i)^2 + d(p',\hat{p}')^2 \tag{3.1}$$

The distance between the measured and true reconstructed points was computed using Cloud Compare. Cloud Compare is a 3D point cloud processing software that offers various advanced processing algorithms , one of which allows the distance between two clouds to be computed. For each reconstructed point Cloud Compare searches for its nearest point in the true cloud and completes the euclidean distance.

The absolute distance between reconstructed points was measured as the progressive reconstruction algorithms purpose is for practical use i.e a best practice combination that results in a low relative distance yet high absolute distance is of no use practically.

The accuracy of reconstruction depends on the accuracy of the determined camera positions and orientations, which furthermore depend on the accuracy of point correspondences.

The accuracy of the camera positions were quantified by measuring the distance between the measured and true camera positions. By measuring the accuracy of the camera positions the drift in cameras throughout the reconstruction was observed for each combination of best practice algorithms.

The accuracy of the camera orientations were not quantified as it could be inferred from observing the relationship between the accuracy of the camera positions and the accuracy of the reconstructed points. For example if the distance between measured and true reconstructed points is large and the distance between measured and true camera positions is small then the error is likely to be in the camera orientations. This is however highly unlikely as due to the progressive nature of the algorithm poor camera orientations should result in subsequent poor camera positions. Furthermore in order to quantify the accuracy of camera orientations the rotation matrix needs to be decomposed into Euler angles which can be a numerically unstable process and lead to inaccurate values.

The accuracy of point correspondences was guaranteed as RANSAC was used to refine matches, however the accuracy of each best practice matching algorithm was determined by comparing the amount of points before and after refinement.

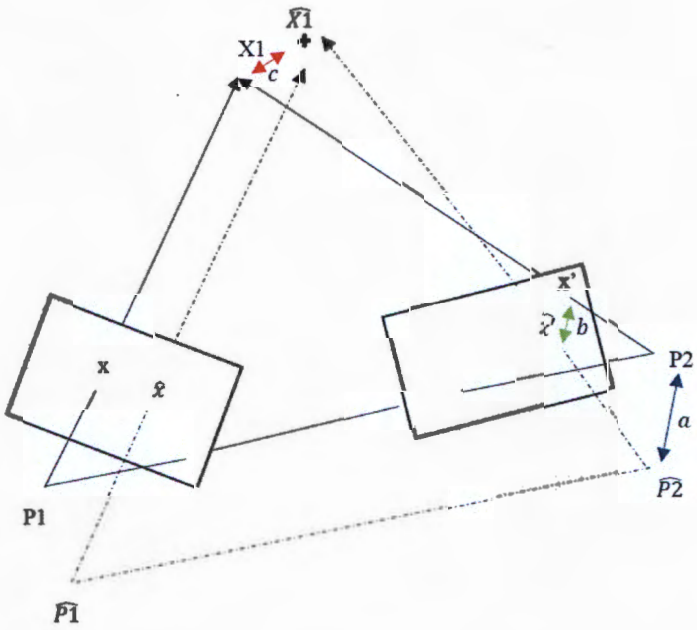


FIGURE 3.12: The camera position error (a) , the reprojection error (b) and the reconstructed point error (c) for a stereo pair of frames. The dotted lines connect the measured camera positions, image points and reconstructed points. As can be seen they do not intersect and an approximate reconstructed point is determined.

3.4.2 Efficiency

The efficiency of the algorithm was broken up into computational efficiency and storage efficiency. Computational efficiency was quantified by time and storage efficiency was quantified by the number of reconstructed points. The computational efficiency of the algorithm as a whole and at each key step was determined (in order to see which steps were the most intensive).

If the purpose of the algorithm is to track the position of the camera then the amount of reconstructed points does not matter and the best practice combination with a high storage efficiency is desired, however if the purpose is for visualization the opposite is true.

Chapter 4

Results

4.1 Introduction

This chapter begins by presenting the data, hardware and software libraries used. Thereafter the results of the empirical analysis i.e. the accuracy of reconstruction, for different combinations of best practice algorithms, is presented using various measures of accuracy before being analysed, the same is repeated for efficiency. The measures of accuracy and efficiency used are presented at the end of the previous chapter. They can be seen below in figure 4.1.

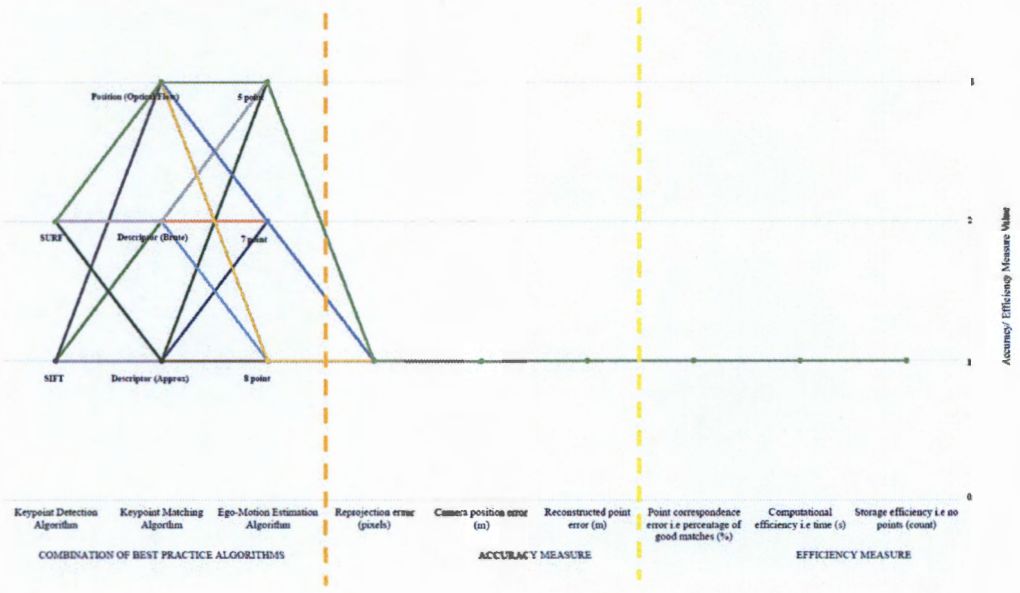


FIGURE 4.1: The measures quantifying the accuracy and efficiency of the reconstruction and their values to be determined in this chapter (provisionally set at 1).

The goal of this chapter is to determine the combination of best practice algorithms that leads to the most accurate and efficient progressive stereo reconstruction i.e the best practice combination.

Where graphs are presented the results for different combinations of best practice algorithms are color coded based on their keypoint detection algorithm. Combinations which use the SURF keypoint detection algorithm are colored dark blue-purple-sky blue and combinations which use the SIFT keypoint detection algorithm are colored orange-gold-yellow. This order of colors refers to the combinations matching algorithm: position (optical flow) matching, brute force descriptor matching and brute force approximate matching.

4.2 Data Used

A multi view outdoor dataset called *Fountain-P11* was reconstructed (Strecha et al., 2008). The dataset contained 11 frames of a fountain (3072 x 2048 pixels). The positions and orientations of the cameras in the world coordinate system were provided, as well as the intrinsic parameters of the camera. The frames were already corrected for lens distortions. The frames can be seen below in figure 4.2. The cameras trajectory can be defined as side motion in comparison to forward motion.

FIGURE 4.2: The eleven frames (frame 0 to 10) comprising the Fountain-P11 dataset (Strecha et al., 2008).

4.3 Hardware Used

A computer with a 3.00GHz Intel Core i5 CPU (four cores running at 3GHZ) and 16.0GB of RAM was used for processing. The Microsoft Windows Experience Index assesses key system components on a scale of 1 to 8.9 and the processor and RAM scores were 7.4 and 7.6 respectively.

4.4 Software Libraries Used

The programing language used was C++. The libraries OpenCV 3.0, Point Cloud Library (PCL) 1.7.2, Boost 1.57 and Intel Thread Building Blocks(TBB) 4.3 were used.

Prebuilt binaries were used except for OpenCV 3 which was compiled from source to allow for the use of Intel Thread Building Blocks (TBB) and the optional "extra modules" which are home to the patented Scale Invariant Feature Transform (SIFT) and Speeded Up Robust Features (SURF) algorithms.

The OpenCV libraries were used for camera calibration and reconstruction as OpenCV is widely used and is an accepted standard, the PCL libraries were used to visualize the reconstructed points , the Boost libraries were used for threading so that the frame acquisition and reconstruction processes could run in parallel and thus allow near real-time reconstruction if frames were acquired from a web camera and lastly the Intel TBB libraries were used to store the frames in a concurrent vector in order to allow both acquisition and reconstruction threads concurrent access. The OpenCV functions used can be seen below in table 4.1.

TABLE 4.1: OpenCV Functions

Task	OpenCV Function
Camera Calibration	calibrateCamera
Frame Undistortion	undistort
Keypoint Detection	detect and compute
Keypoint Matching	match/calcPosition(OpticalFlow)PyrLK
Fundamental Matrix Computation	findFundamentalMat
Ego-Motion Estimation	recoverPose/solvePNPRANSAC
Combining Rotation and Translation	composeRT
Keypoint Reconstruction	correctMatches and triangulatePoint.

4.5 Accuracy Results

The accuracy of reconstruction was quantified by the reprojection error, the distance between the measured and true camera positions, the distance between the measured and true reconstructed points and the ratio of good to bad point correspondences, which in actuality has no effect on accuracy as RANSAC was used.

4.5.1 Reprojection Error

For each combination of best practice algorithms the average and standard deviation of the reprojection error for all reconstructed points was determined. The results can be seen below in table 4.2. For all combinations of best practice algorithms, excluding *SURF - Descriptor(Approx) - 8-point* and *SURF - Descriptor(Approx) - 7-point* which have gross errors, the average reprojection error was 0.66 pixels and the average standard deviation (population) was 0.4 pixels.

The combination of best practice algorithms *SIFT - Descriptor(Brute) - 8-point* had the lowest mean reprojection error with a value of 0.148 pixels and a standard deviation of 0.047 pixels.

A bar graph of the mean and standard deviation of the reprojection errors for each combination of best practice algorithms can be seen in figure 4.3. The combinations of best practice algorithms are on the horizontal axis and the error values are on the vertical axis. The influence of restricting the refined matches to within 10 percent of the image boarders in order to reduce the parallelism of rays can be seen by comparing the bar graph to the line graph. The line graph represents the reprojection errors if all refined matches were reconstructed and is clearly higher (above the bar graph) for all combinations of best practice algorithms. For all combinations excluding *SURF - Descriptor(Approx) - 8-point* and *SURF - Descriptor(Approx) - 7-point*, the average reprojection error for all refined matches was 0.78 pixels and the average standard deviation was 0.52 pixels, this is considerably higher than the values of 0.66 and 0.4 pixels attained when the refined matches were bounded.

TABLE 4.2: Mean and standard deviation of the reprojection error for each combination of best practice algorithms

			Reprojection Error (pixels)			
			Bound		No Bound	
Keypoint	**Matching**	**Ego-Motion**	**Mean**	**Stdev**	**Mean**	**Stdev**
SURF	Optical	8-point	0.335	0.295	0.367	0.334
		7-point	0.409	0.216	0.493	0.298
		5-point	0.291	0.199	0.353	0.339
	Brute	8-point	0.273	0.144	0.316	0.162
		7-point	0.763	0.327	0.886	0.429
		5-point	0.380	0.181	0.418	0.191
	Approx	8-point	41.772	41.264	39.858	34.794
		7-point	8.968	11.644	10.339	13.111
		5-point	0.628	0.381	0.675	0.408
SIFT	Optical	8-point	0.851	0.391	1.001	0.433
		7-point	0.303	0.106	0.423	0.309
		5-point	0.399	0.190	0.495	0.240
	Brute	8-point	0.148	0.047	0.177	0.061
		7-point	1.545	0.736	1.760	0.910
		5-point	0.448	0.213	0.505	0.250
	Approx	8-point	3.097	2.664	3.861	3.626
		7-point	0.443	0.193	0.532	0.227
		5-point	0.183	0.094	0.205	0.101

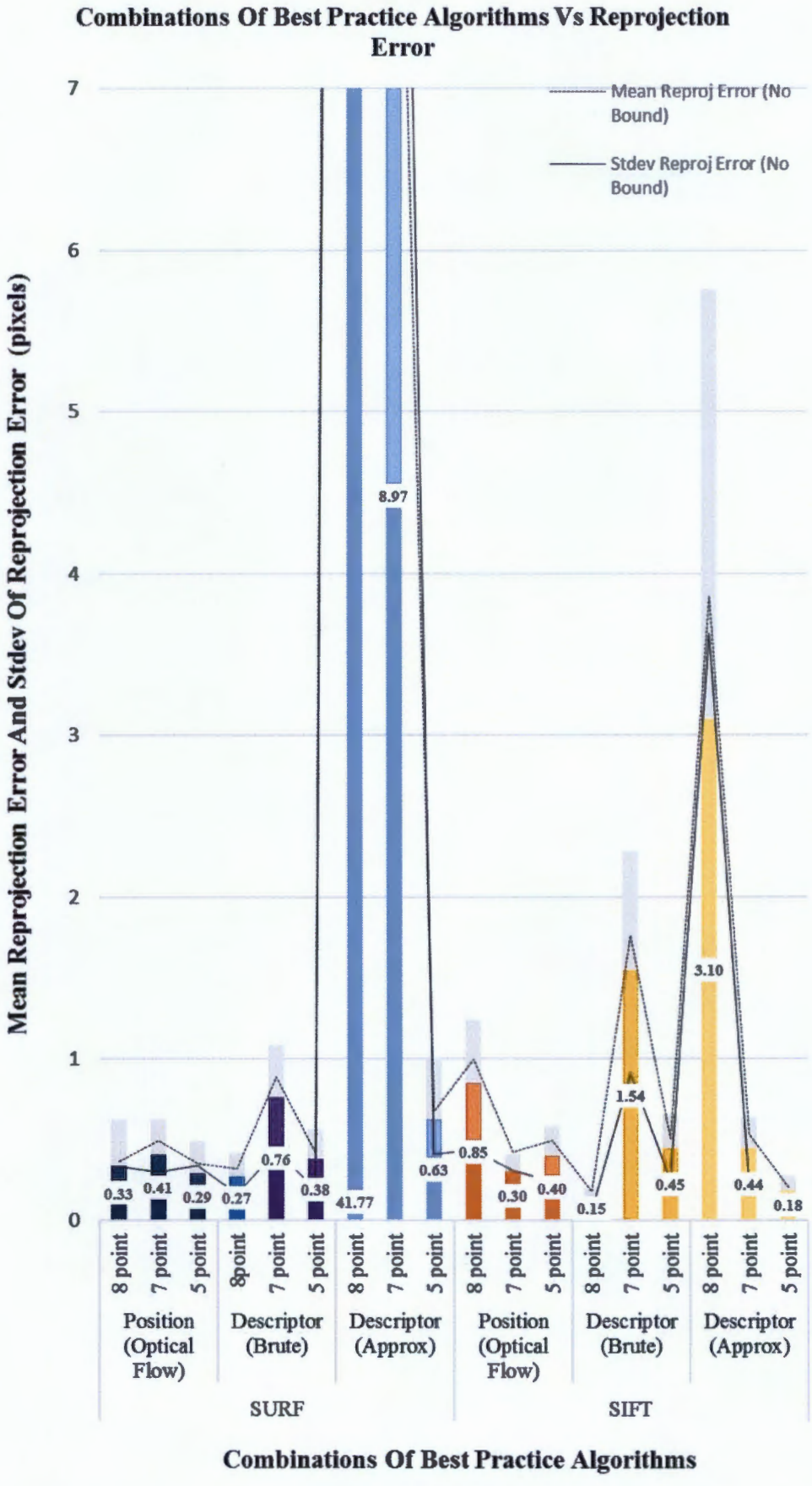


FIGURE 4.3: Graph of the mean and standard deviation of the reprojection error for each combination of best practice algorithms

4.5.2 Camera Position Error

For each combination of best practice algorithms the average and standard deviation of the distance from the true camera positions to the measured camera positions was determined. The results can be seen below in table 4.3. For all combinations excluding *SURF - Descriptor(Approx) - 8-point* and *SURF - Descriptor(Approx) - 7-point* which have gross errors, the average camera position error was 0.17m and the average standard deviation (population) was 0.08m.

The combination of best practice algorithms *SIFT - Descriptor(Approx) - 5-point* had the lowest mean camera position error with a value of 0.011m and a standard deviation of 0.003m.

A bar graph of the mean and standard deviation of the camera position errors for each combination of best practice algorithms can be seen in figure 4.4. The combinations of best practice algorithms are on the horizontal axis and the error values are on the vertical axis.

For each combination of best practice algorithms the error in the determined camera positions at each frame was also determined. A line graph of the camera position error at each frame for each combination of best practice algorithms can be seen in figure 4.5. The frame numbers are on the horizontal axis and the error values are on the vertical axis. This graph is a good proxy to determine whether the distance between the measured and true camera positions increases over frames i.e whether the cameras drift. Visually the combination of best practice algorithms *SIFT-Descriptor(Approx)-5* point maintained the lowest accuracy throughout and did not appear to drift.

TABLE 4.3: Mean and standard deviation of the error in the camera positions for each combination of best practice algorithms

			Distance (m)	
Keypoint	**Matching**	**Ego-Motion**	**Mean**	**Stdev**
SURF	Optical	8-point	0.075	0.033
		7-point	0.090	0.033
		5-point	0.119	0.093
	Brute	8point	0.085	0.038
		7-point	0.130	0.044
		5-point	0.122	0.048
	Approx	8-point	7.527	6.293
		7-point	2.577	1.191
		5-point	0.337	0.194
SIFT	Optical	8-point	0.180	0.098
		7-point	0.122	0.065
		5-point	0.144	0.095
	Brute	8-point	0.031	0.010
		7-point	0.424	0.179
		5-point	0.189	0.088
	Approx	8-point	0.552	0.228
		7-point	0.179	0.073
		5-point	0.011	0.003

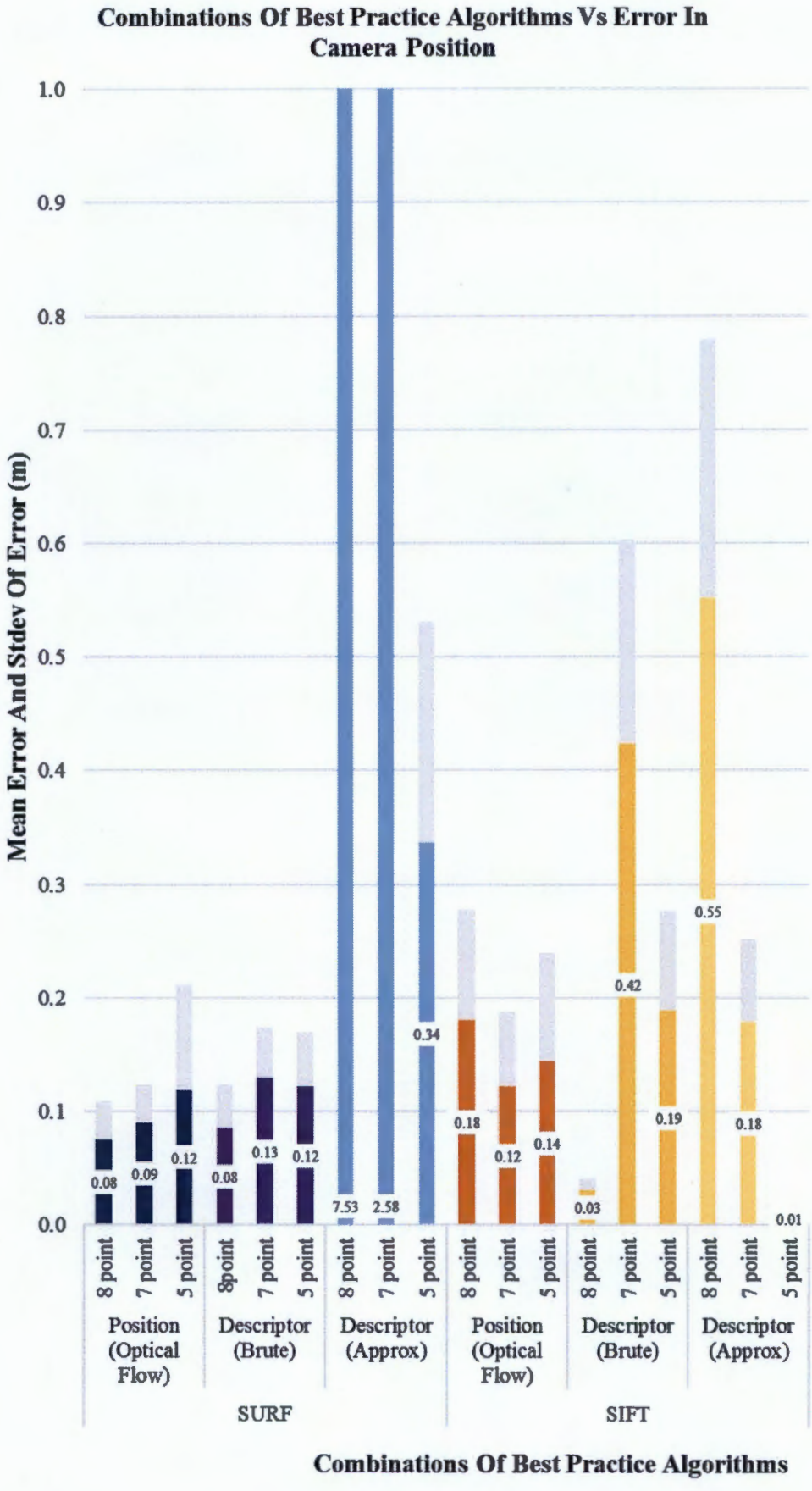


FIGURE 4.4: Graph of the mean and standard deviation of the error in camera position for each combination of best practice algorithms

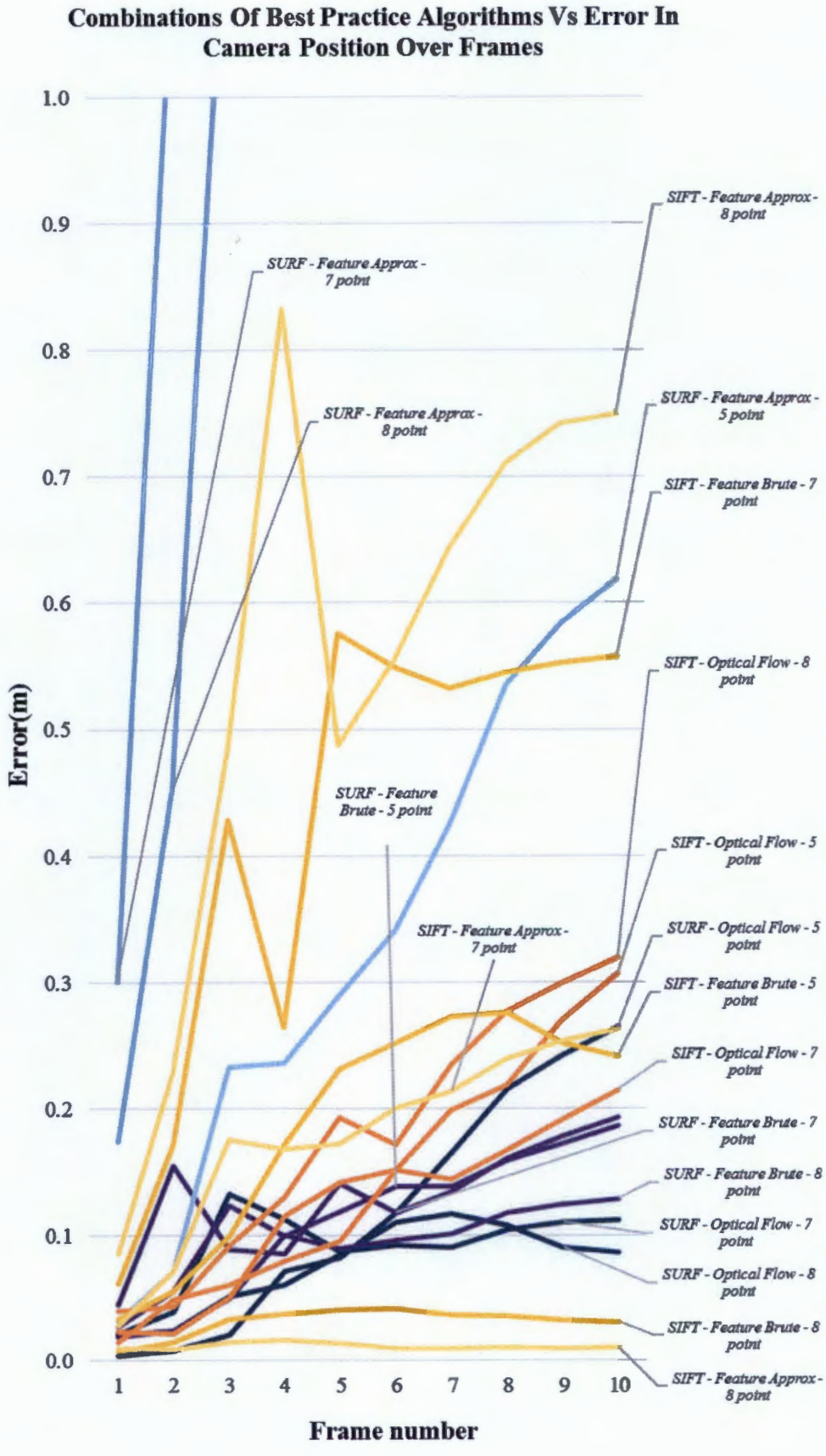


FIGURE 4.5: Graph of the error in camera position at each frame for each combination of best practice algorithms. In the dataset *Fountain-P11* there are 11 frames (numbered 0 to 10) and the error is computed for frame 1 to 10.

4.5.3 Reconstructed Point Error

For each combination of best practice algorithms the average and standard deviation of the distance from the true points to the reconstructed points was determined. The results can be seen below in table 4.4. For all combinations of best practice algorithms, excluding *SURF - Descriptor(Approx) - 8-point* and *SURF - Descriptor(Approx) - 7-point* which have gross errors, the average reconstructed point error was 0.14m and the average standard deviation (population) was 0.18m. The standard deviation was larger as the reconstructed point cloud was noisy because points with high reprojection errors were not removed.

The best practice combination *SURF - OpticalFlow - 7-point* had the lowest mean camera position error with a value of 0.044m and a standard deviation of 0.048m.

A bar graph of the mean and standard deviation of the errors in the reconstructed points for each combination of best practice algorithms can be seen in figure 4.6. The combinations of best practice algorithms are on the horizontal axis and the error values are on the vertical axis

TABLE 4.4: Mean and standard deviation of the error in the reconstructed points for each combination of best practice algorithms

			Distance (m)	
Keypoint	**Matching**	**Ego-Motion**	**Mean**	**Stdev**
SURF	Optical	8-point	0.049	0.046
		7-point	0.044	0.048
		5-point	0.107	0.079
	Brute	8point	0.074	0.157
		7-point	0.119	0.182
		5-point	0.129	0.193
	Approx	8-point	0.695	0.343
		7-point	0.575	0.296
		5-point	0.252	0.187
SIFT	Optical	8-point	0.089	0.055
		7-point	0.059	0.051
		5-point	0.154	0.063
	Brute	8-point	0.132	0.308
		7-point	0.254	0.293
		5-point	0.156	0.305
	Approx	8-point	0.329	0.317
		7-point	0.180	0.273
		5-point	0.137	0.315

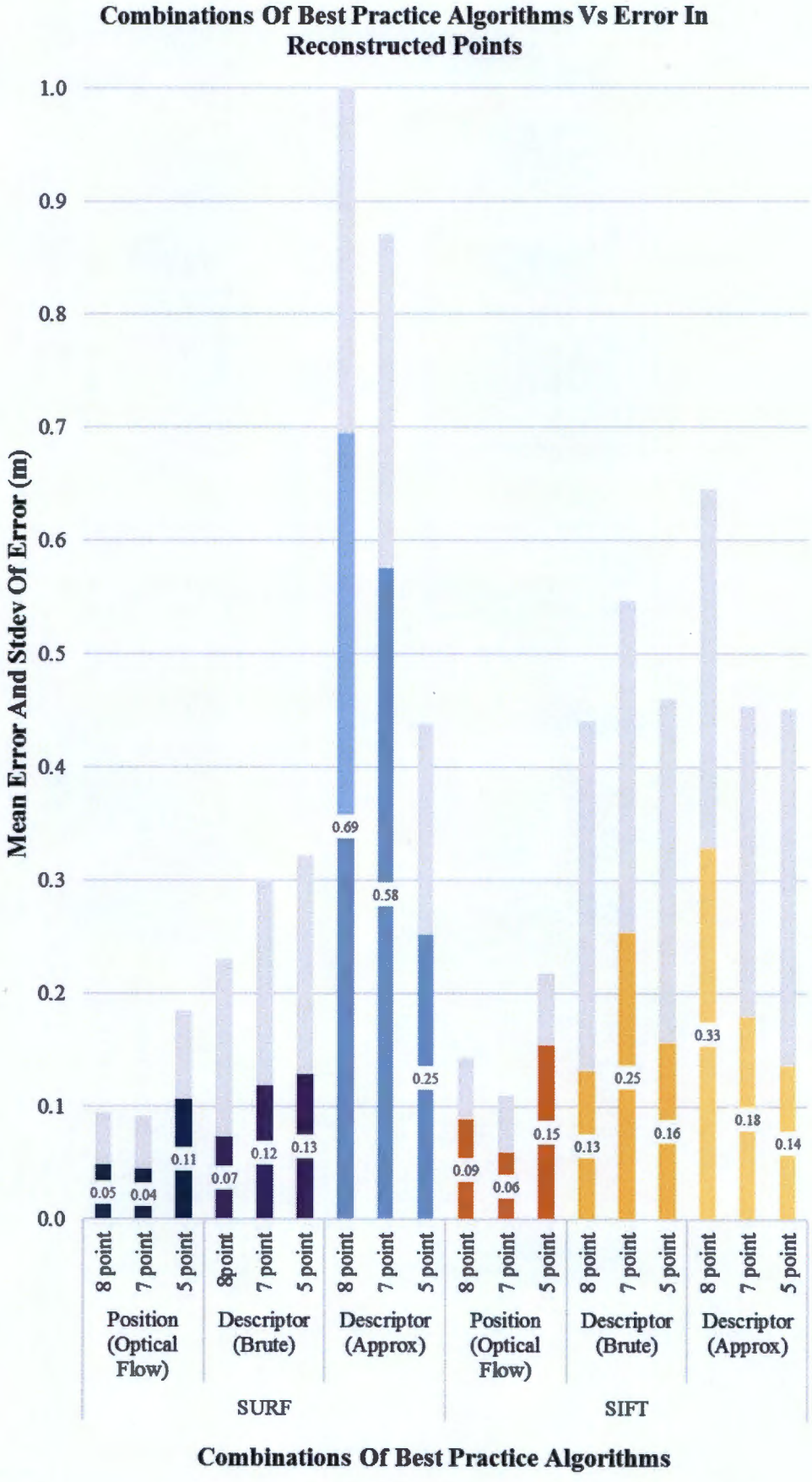


FIGURE 4.6: Graph of the mean and standard deviation of the error in reconstructed points for each combination of best practice algorithms

4.5.4 Point Correspondence Error

For each combination of best practice algorithms the percentage of matches which were good were determined. The results can be seen below in table 4.5.

The combination of best practice algorithms *SURF - Desciptor(Approx) - 7-point* and *SURF - Desciptor(Approx) - 8-point* had the most amount of matches removed, with only 33 percent of matches being preserved i.e good matches. The combination of best practice algorithms *SIFT - Position(OpticalFlow) - 5-point* had the least amount of matches removed with 77 percent of matches being preserved.

A bar graph of the percentage of matches preserved for each combination of best practice algorithms can be seen in figure 4.7. The combinations of best practice algorithms are on the horizontal axis and the percentages are on the vertical axis

As RANSAC was used only accurate correspondences i.e good matches were used. Therefore erroneousness point correspondences had no effect on the reconstruction. There will however always be errors in corresponding points due to limitations of the image resolution.

TABLE 4.5: Percentage of matches preserved for each combination of best practice algorithms

			Matches (count)		
Keypoint	**Matching**	**Ego-motion**	**All**	**Refined**	**Percentage (%)**
SURF	Optical	8-point	3359	1488	44.299
		7-point	3359	1488	44.299
		5-point	3359	1803	53.677
	Brute	8-point	13913	7711	55.423
		7-point	13913	7711	55.423
		5-point	13913	8377	60.210
	Approx	8-point	20950	6973	33.284
		7-point	20950	6973	33.284
		5-point	20950	8076	38.549
SIFT	Optical	8-point	491	328	66.802
		7-point	491	328	66.802
		5-point	491	378	76.986
	Brute	8-point	2628	1635	62.215
		7-point	2628	1635	62.215
		5-point	2628	1634	62.177
	Approx	8-point	3768	1538	40.817
		7-point	3768	1538	40.817
		5-point	3768	1686	44.745

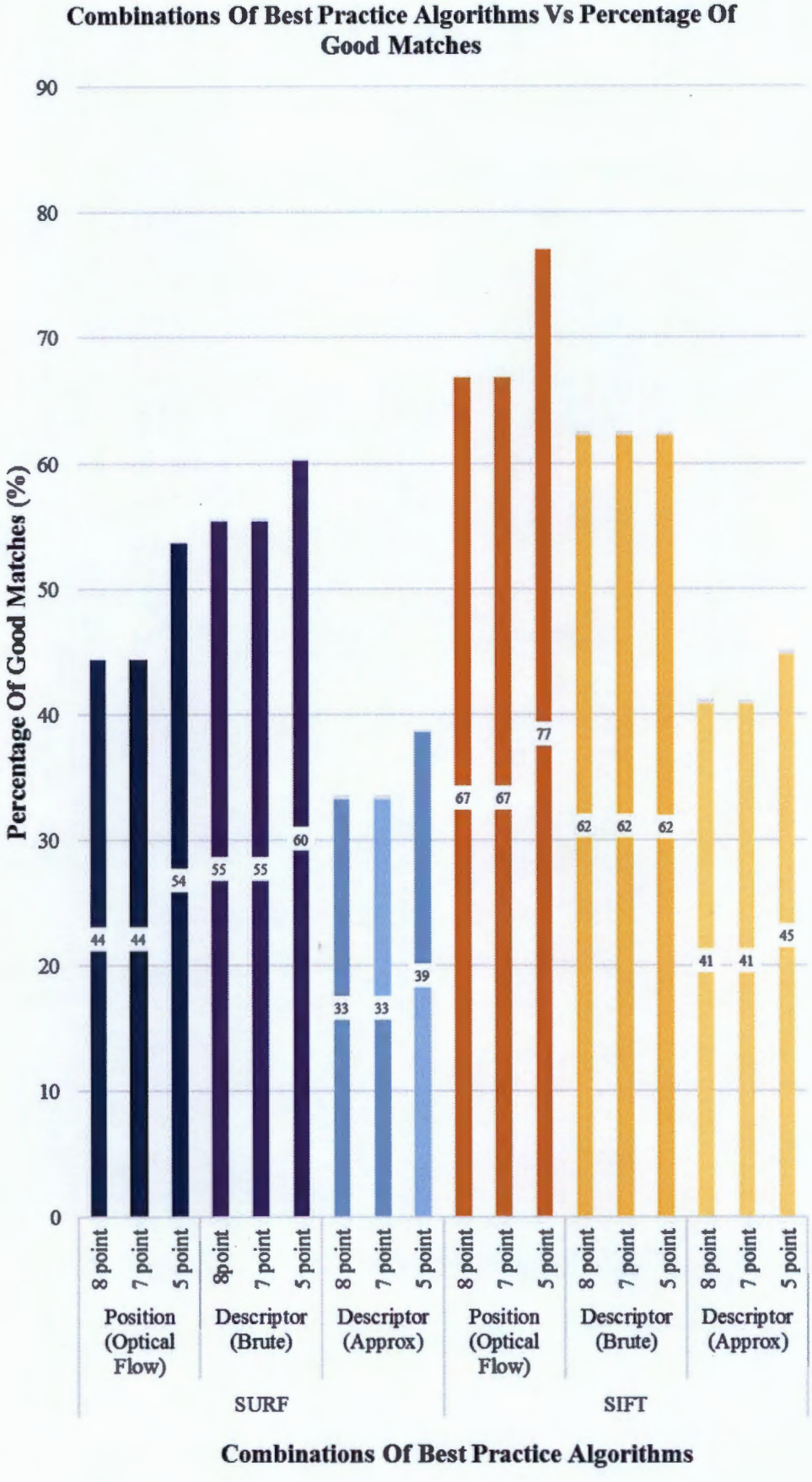


FIGURE 4.7: Graph of the percentage of good matches for each combination of best practice algorithms

4.5.5 Analysis Of Accuracy

Firstly the influences each key step and subsequent best practice algorithms chosen for that step have on the accuracy of the reconstruction are analyzed and thereafter the combination of best practice algorithms that leads to the most accurate reconstruction is determined and presented.

4.5.5.1 Key Steps

Keypoint detection and matching:

The choice of the keypoint detection algorithm (SIFT/SURF) and keypoint matching algorithm (Descriptor(Brute/Approx)/ Position(Optical Flow)) has no effect on the accuracy of the reconstruction as RANSAC is used to refine matches and ensure accurate point correspondences.

Ego-motion estimation:

To recap, for the first pair of frames the ego-motion is determined by decomposing the fundamental matrix and for sequential frames it is determined from 3D-2D correspondences in order to ensure the same scale. The fundamental matrix is still decomposed however the rotation and translation values are used as provisionals, therefore if they are inaccurate their influence is minimal.

Therefore the accuracy of ego-motion depends on the accuracy of the fundamental matrix for the first pair of frames and the accuracy of the 3D and 2D point correspondences for sequential frames. Throughout the reconstruction algorithm accurate correspondences are ensured by RANSAC and therefore the major concern is the accuracy of the reconstructed points, and thus the accuracy of the ego-motion determined for the first pair of frames. The accuracy of the reconstructed points however not only depends on the accuracy of the determined camera positions but also depends on the intersection angle of camera rays, by restricting the refined matches to within 10 percent of the image boarders the parallelism of rays is reduced and the accuracy of reconstruction is improved. This can be seen by observing the reprojection errors in figure 4.3. The line graph represents the errors if the refined matches were not restricted.

In order to determine the fundamental matrix a random selection of 5,7 or 8-points is used, depending on the ego-motion estimation algorithm. In order for the fundamental matrix to be accurate it is desired that the selection of points do not contain noisy points and do not lie on a plane (degenerate configuration), as here the matrix determined will

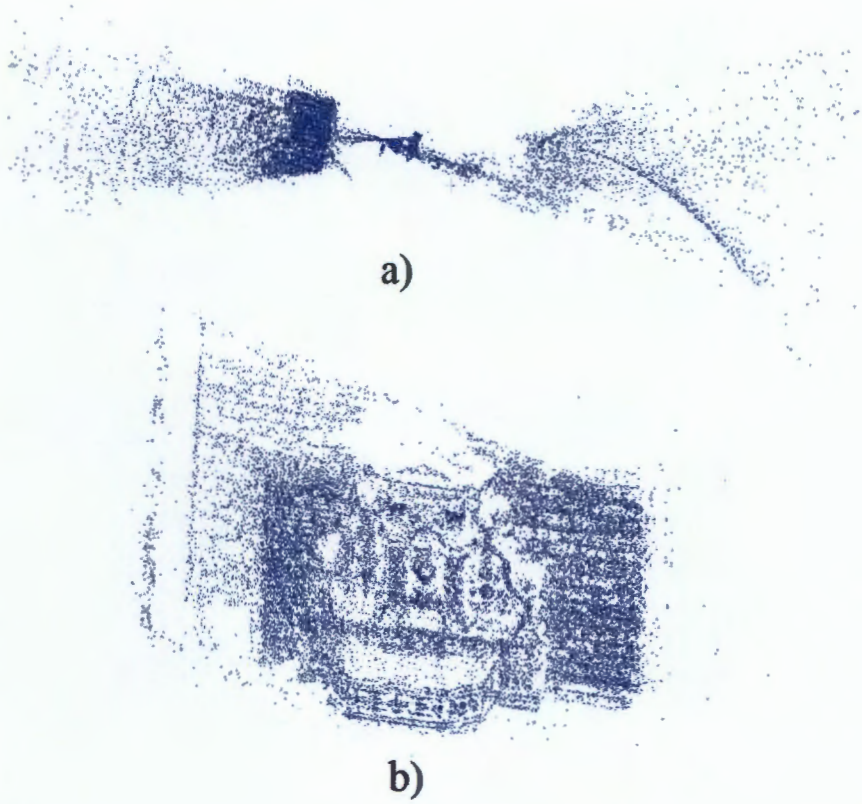


FIGURE 4.8: Reconstruction using the combination of best practice algorithms *SURF - Descriptor(Approx) - 7-point* (a) and *SURF - Descriptor(Approx) - 5-point* (b). The reconstruction using the 7-point algorithm is inaccurate as the ego-motion for the first pair of frames was computed from a planar set of points. The reconstruction using the 5-point algorithm is more accurate as the 5-point algorithm is robust to a planar selections of points (degenerate configuration).

represent more of a homography and will not encapsulate the epipolar geometry of the two cameras correctly.

Therefore the gross errors of the combination of best practice algorithms *SURF - Descriptor(Approx) - 8-point* and *SURF - Descriptor(Approx) - 7-point* are the result of the ego-motion for the initial pair of frames being determined from a fundamental matrix which has been determined from a selection of planar points (degenerate configuration). The 7-point algorithm results in three fundamental matrices if the point configuration is degenerate and the 8-point algorithm simply results in an inaccurate matrix. The 5-point algorithm however is robust to degenerate configurations, this can be seen in figure 4.8. The variation of accuracy among other combinations of best practice algorithms can be accredited to pixel noise.

Overall the influences of the choice of ego-motion algorithm vary more for combinations of best practice algorithms which use SIFT. This however has nothing to do with SIFT

being chosen, it is simply due to the fortunate lower pixel noise in the random selection of point correspondences used to define the fundamental matrix and ego-motion for the first pair of frames.

It must be noted that the cameras trajectory in the dataset can be classified as side motion and not forward motion. For such case the choice of ego-motion estimation has a minimal effect on the accuracy, however it is known that the 8-point algorithm is more accurate than the 5-point algorithm for forward motion. The ego-motion estimated for sequential frames is done using 3D-2D correspondences which is however also not robust to forward motion. Therefore regardless of the choice of ego-motion algorithm for the first pair of frames forward motion is not advised. For side motion the choice of the ego-motion estimation algorithm (8-point/7-point/5-point) has no influence on the accuracy of the reconstruction, however the selection of points used to estimate it do and the 5-point algorithm is robust to a degenerate selection.

Furthermore due to the sequential nature of the algorithm inaccuracies have a compounded affect and a drift in errors can be observed, however the novel 3D-2D resection approach used in the progressive reconstruction algorithm aims to reduce this by not re-reconstructing duplicate points. The source of inaccuracies for all combinations of best practice algorithms can be seen by observing the errors in the camera position at the first frame, this can be seen in figure 4.5.

As mentioned accurate correspondences are ensured by RANSAC, however brute force matching is inherently more accurate (less matches are removed) than approximate or optical flow matching. Brute force is more accurate than approximate as every descriptor is matched and matches are only made if they are the best match in both directions. Position (optical flow) matching inherently is the most accurate, however if the correspondences are dense there may be more than one close nearest neighbor to the optical determined position and the risk of an incorrect match is often too high and so a match is not made, for that reason the percentage of good matches is lower. This can be seen in graph 4.7 where SURF has more points and thus a higher percentage of matches are removed than compared to SIFT when position (optical flow) matching is chosen.

4.5.5.2 The Best Practice Combination

In order to determine the combination of best practice algorithms that results in the most accurate reconstruction the reprojection errors, errors in camera position and errors in reconstructed points were examined.

From looking at the graphs in figures 4.3, 4.5 and 4.6 it is clear that there is a correlation between the error measures. However certain measures are more faithful representations

of accuracy than others, for example due to the the real time nature of the algorithm points with large reprojection errors are not removed and thus the mean and standard deviation of the reprojection errors are not guaranteed to be representative of the actual accuracy, furthermore such outlier points have the same effect on errors in the reconstructed points therefore also reducing the representativeness of this measure. The most faithful measure of accuracy is therefore the camera positions as RANSAC is used to ensure ego-motion estimation is free from outliers. The combination of best practice algorithms with the lowest camera position error is *SIFT - Descriptor(Approx) - 5-point.*

4.6 Efficiency Results

The computational and storage efficiency of reconstruction was quantified by time and the number of reconstructed points respectively.

4.6.1 Computational Efficiency

For each combination of best practice algorithms the time of computation was recorded. The results can be seen below in table 4.6.

The combinations of best practice algorithms which used SIFT and position (optical flow) matching had the same computational efficiency and were the most computationally efficient. The combination of best practice algorithms *SURF - Descriptor(Approx) - 5-point* was the least.

A bar graph of the time of computation for each combination of best practice algorithms can be seen in figure 4.9. The combinations of best practice algorithms are on the horizontal axis and the times are on the vertical axis. There is a clear distinction between the combinations of best practice algorithms which use SIFT and SURF with combinations that use SIFT being approximately three times more computationally efficient than SURF. The reason for this is because SURF results in more keypoints and thus more computational power is needed for matching and reconstruction compared to SIFT.

In terms of keypoint detection however SURF is three times more computationally efficient than SIFT, in terms of matching the order of computational efficiency is, in increasing efficiency: descriptor (approx), descriptor(brute) and then position (optical flow). In terms of ego-motion estimation the order of increasing efficiency is 5-point, 7-point and 8-point. This can be see below in table 4.7.

A bar graph of the time of computation at each key step for each best practice algorithm can be seen in figure 4.10. The key steps are on the horizontal axis and the times are on the vertical axis.

TABLE 4.6: Time of computation for each combination of best practice algorithms

Keypoint	Matching	Ego-Motion	Time (s)
SURF	Optical	8-point	700
		7-point	704
		5-point	867
	Brute	8point	2864
		7-point	2936
		5-point	3367
	Approx	8-point	2712
		7-point	2763
		5-point	4568
SIFT	Optical	8-point	182
		7-point	183
		5-point	187
	Brute	8-point	503
		7-point	515
		5-point	520
	Approx	8-point	553
		7-point	561
		5-point	717

TABLE 4.7: Time of computation of each key step for varying best practice algorithms

		Time (s)	
		SURF	**SIFT**
Detection		7.111	21.778
Matching	Optical	13.000	1.000
	Brute	57.667	23.333
	Approx	65.333	27.000
Ego-Motion	8-point	3.667	0.333
	7-point	7.333	1.667
	5-point	34.000	3.667

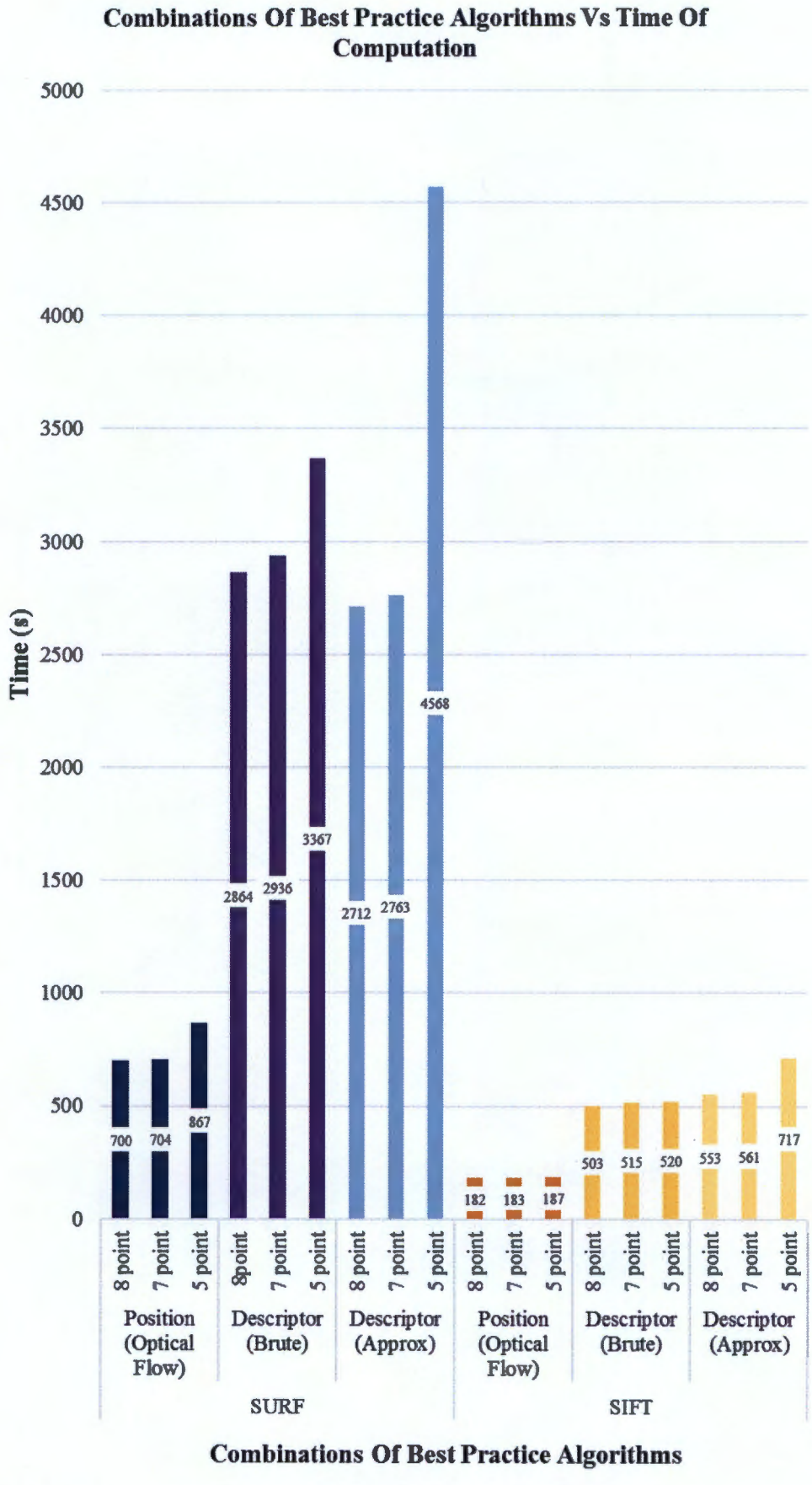


FIGURE 4.9: Graph of the computation time for each combination of best practice algorithms

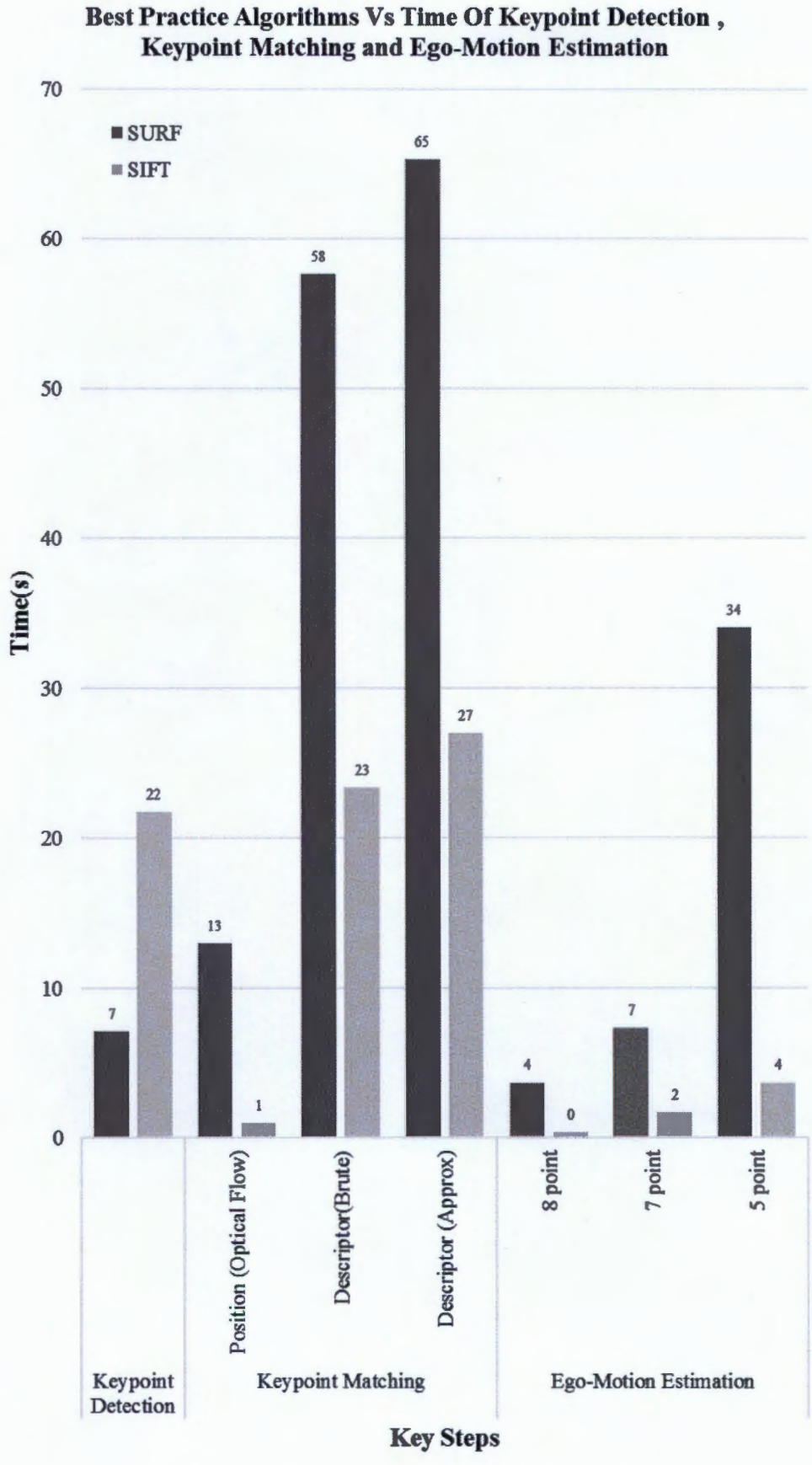


FIGURE 4.10: Graph of the time of keypoint detection, keypoint matching and ego-motion estimation for varying best practice algorithms

4.6.2 Storage Efficiency

For each combination of best practice algorithms the number of points reconstructed was recorded. The results can be seen below in table 4.8.

The combinations of best practice algorithms which used SIFT and position (optical flow) matching had the same storage efficiency and were the most storage efficient. The combination of best practice algorithms *SURF - Descriptor(Approx) - 5-point* was the least.

A bar graph of the number of points reconstructed for each combination of best practice algorithms can be seen in figure 4.11. The combinations of best practice algorithms are on the horizontal axis and the number of reconstructed points are on the vertical axis.

There is a clear distinction between combinations of best practice algorithms using SIFT and SURF with SIFT being approximately three times more storage efficient. In terms of matching the order of storage efficiency is, in increasing efficiency: descriptor (approx), descriptor(brute) and position (optical flow). In terms of ego-motion estimation the order of increasing storage efficiency is 5-point, 7-point and 8-point.

TABLE 4.8: Number of points reconstructed for each combination of best practice algorithms

Keypoint	Matching	Ego-Motion	Points
SURF	Optical	8-point	19565
		7-point	19566
		5-point	22029
	Brute	8-point	32830
		7-point	32827
		5-point	35476
	Approx	8-point	32407
		7-point	32404
		5-point	38641
SIFT	Optical	8-point	2919
		7-point	2919
		5-point	3113
	Brute	8-point	6940
		7-point	6940
		5-point	7127
	Approx	8-point	6350
		7-point	6351
		5-point	7226

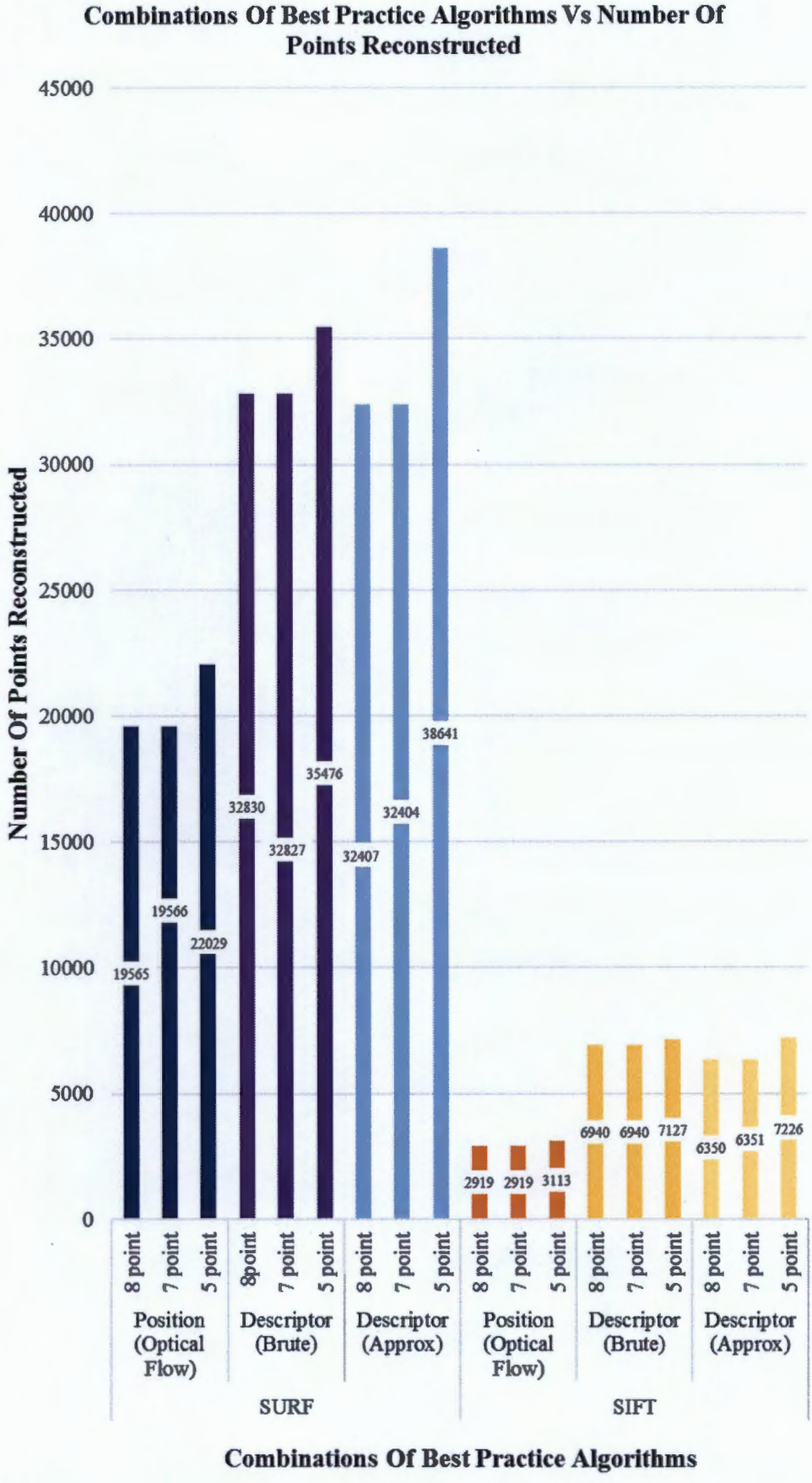


FIGURE 4.11: Graph of the number of points reconstructed for each combination of best practice algorithms

4.6.3 Analysis Of Efficiency

Firstly the influences each key step and subsequent best practice algorithms chosen for that step have on the efficiency of the reconstruction are analyzed and thereafter the combination of best practice algorithms that leads to the most efficient reconstruction is determined and presented.

4.6.3.1 Key Steps

Keypoint detection:

The choice of the keypoint detection algorithm (SIFT/SURF) has a big influence on the computational and storage efficiency of the reconstruction. SURF results in significantly more reconstructed points, however in significantly more time. Therefore the rate of points reconstructed per second is similar and computational and storage efficiency are linearly dependent and therefore can both be referred to under the same umbrella of "efficiency".

Keypoint matching:

Matching by position (optical flow) is significantly more efficient than descriptor based matching. Brute force descriptor matching is more efficient than approximate matching. This can be seen in figure 4.9 and figure 4.11. This is because of the high dimensionality of the SURF descriptor which makes creating a tree diagram for approximate searching computationally exhaustive.

Ego-motion estimation:

In terms of the ego-motion algorithm (8-point,7-point,5-point) the efficiency of algorithms are in the following order of increasing efficiency: 5-point, 7-point and 8-point. The 8-point algorithm when SIFT is used executes in zero seconds which is real-time. The relative computational efficiencies can be seen in figure 4.10.

4.6.3.2 The Best Practice Combination

If the purpose of the reconstruction algorithm is for visualization then more points are desired and a combination of best practice algorithms using SURF is recommended, however if the purpose is to track the camera position then the amount of points is not important and a combination using SIFT is recommended.

The combination of best practice algorithms that is best suited to visualization i.e uses SURF and is the most computationally efficient is *SURF - Position(OpticalFlow) - 8-point* and *SURF - Position(OpticalFlow) - 7-point.*

The combination of best practice algorithms that is best suited to camera tracking i.e uses SIFT and is the most computationally efficient is *SIFT - Position(OpticalFlow) - 8-point* and *SIFT - Position(OpticalFlow) - 7-point.*

4.7 Summary Of Results

For progressive stereo reconstruction the computational and storage efficiencies are linearly dependent and are therefore both referred to as efficiency.

For progressive stereo reconstruction the choice of keypoint detection algorithm has no influence on the accuracy of the reconstruction, however it effects the efficiency. Reconstructions for combinations using SURF are 3x less efficient than those that use SIFT and therefore are best suited for visualization purposes whereas combinations which use SIFT are best suited for camera tracking purposes.

The choice of matching algorithm has no effect on the accuracy as RANSAC is used, however the selection has a considerable effect on the efficiency, position (optical flow) matching is considerably more efficient than brute force and approximate descriptor matching.

The choice of the ego-motion algorithm effects the efficiency of the reconstruction and the accuracy of reconstruction for the first pair of frames depending on the selection of points used to determine the ego-motion. The order of efficiency, in increasing efficiency, is 5-point, 7-point and 8-point. The 5-point algorithm is robust to planar degeneracies and therefore is the most reliable. The downside is that it is the most computationally intensive.

Therefore there is no influence on the accuracy between several algorithms for keypoint detection/matching and no significant influence between ego-motion estimation algorithms if the selection of points chosen is not degenerate. The later is because here the differences in accuracy amongst ego-motion algorithms are attributed to pixel noise in the random selection of points (which are used to compute the fundamental matrix and ego-motion thereof) and are therefore not significant. This means that algorithms which improve efficiency can be chosen knowing there will be no significant compromise in accuracy. These are new insights. This was not expected, if it was known certain combinations have no or no significant influence there would have been no need to study the effect of different combinations of best practice algorithms on the accuracy and efficiency of the 3D reconstruction

If the purpose of real time progressive stereo reconstruction is to visualize/reconstruct the scene in near real-time on a mobile device the best practice combination is *SURF - Position(OpticalFlow) - 5-point.* Snapshots of the progressive stereo reconstruction sequence of the fountain dataset using this combination can be seen in figure 4.15. The complete reconstruction of the fountain dataset using this combination can be seen below in figure 4.12.

FIGURE 4.12: Complete reconstruction of the fountain dataset using the best practice combination SURF-Position(OpticalFlow)-5point

If the purpose of real time progressive stereo reconstruction is to track the position of the camera in the scene in near real-time on a mobile device the best practice combination is *SIFT - Position(OpticalFlow) - 5-point.* Snapshots of the progressive stereo reconstruction sequence of the fountain dataset using this combination can be seen in figure 4.16. The complete reconstruction of the fountain dataset using this combination can be seen below in figure 4.13.

FIGURE 4.13: Complete reconstruction of the fountain dataset using the best practice combination - SIFT-Position(OpticalFlow)-5point

For comparison the fountain dataset was sparsely reconstructed using Agisoft PhotoScan 1.2.4 in default settings (medium resolution). A snapshot of the reconstruction can be seen below in figure 4.14. The reconstruction took 247 seconds - significantly slower than the 187 seconds the progressive stereo reconstruction algorithm took when the best practice combination SIFT-Position(OpticalFlow)-5point was used.

FIGURE 4.14: Complete reconstruction of the fountain dataset using Agisoft Photoscan - Medium Resolution - Sparse Point Cloud

As position (optical flow) matching is used the requirement of using a keypoint detection algorithm such as SIFT and SURF that is invariant to scale and rotation is not relaxed because it is important the keypoints detected do not lie on a planar surface as in this case the accuracy of the ego-motion estimated for sequential views may be impaired.

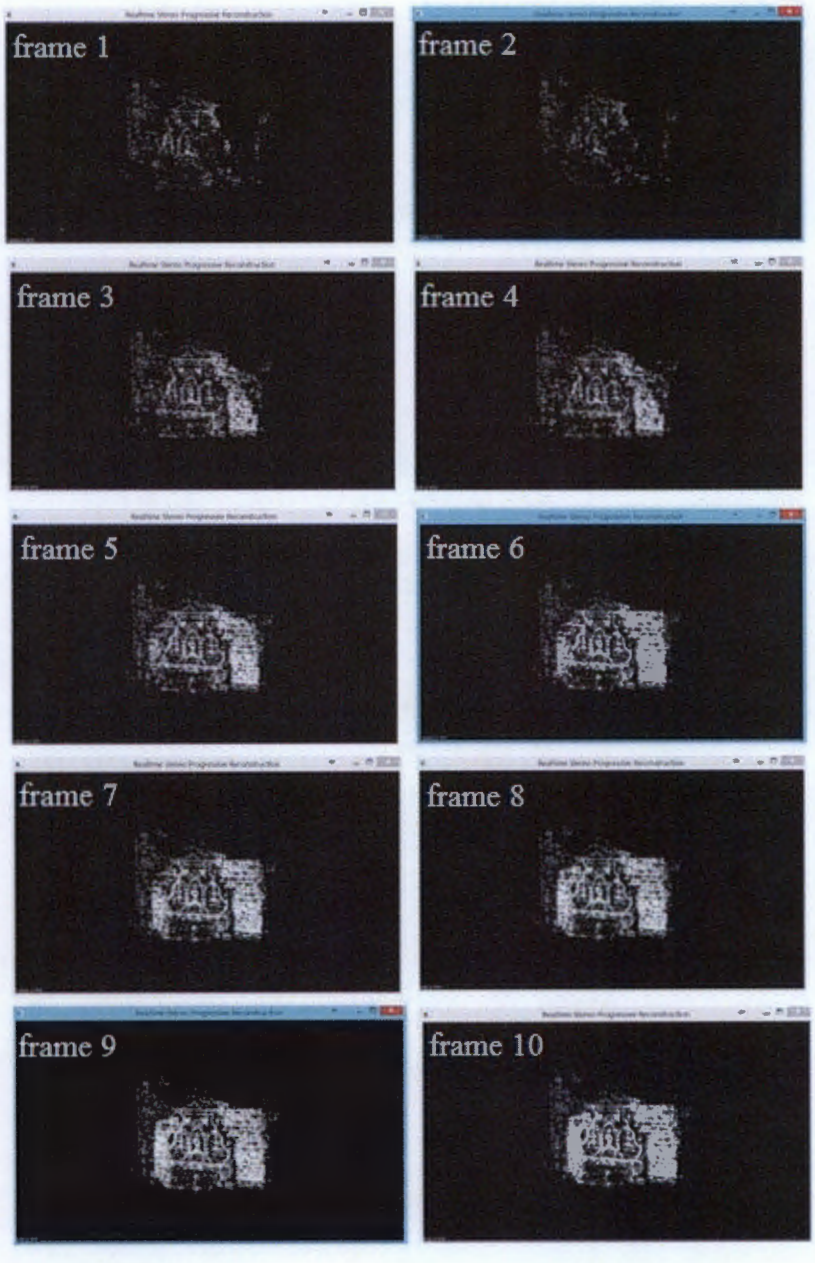


FIGURE 4.15: Progressive stereo reconstruction sequence of the fountain dataset using the best practice combination *SURF-Position(OpticalFlow)-5-point.*

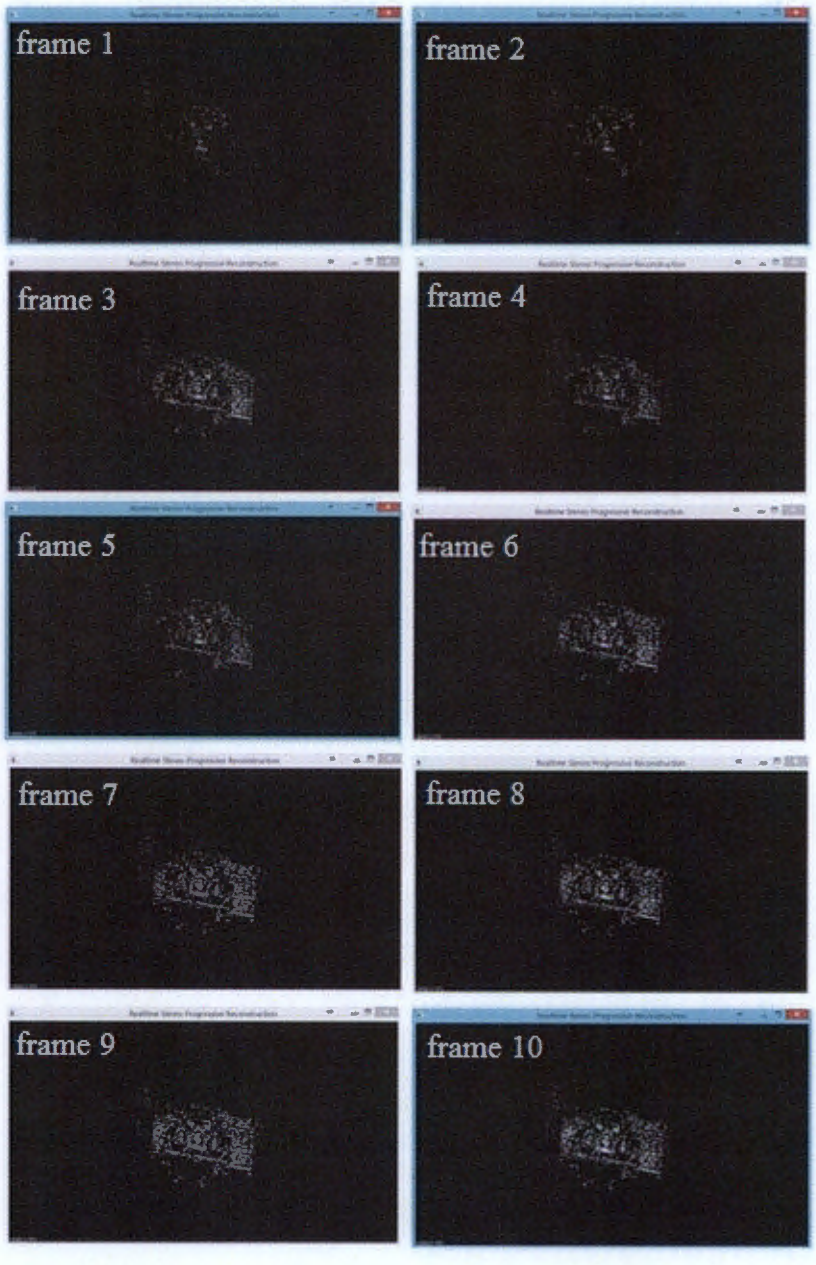


FIGURE 4.16: Progressive stereo reconstruction sequence of the fountain dataset using the best practice combination *SIFT-Position(OpticalFlow)-5-point*.

Chapter 5

Conclusion

A novel progressive stereo reconstruction algorithm has been presented that is robust to drift and does not require computationally exhaustive post processing algorithms such as the bundle adjustment to ensure accuracy.

The best practice algorithms chosen for the key steps of progressive stereo reconstruction: keypoint detection, keypoint matching and ego-motion estimation were found to have no influence on the accuracy of reconstruction, however they were found to influence the efficiency of reconstruction. The selection and configuration of points used to compute the ego-motion for the first pair of frames was however found to influence the accuracy of reconstruction.

The combination of best practice algorithms that leads to the most accurate and efficient progressive stereo reconstruction was found i.e the best practice combination. The combination differs depending on the purpose of the progressive reconstruction. If the purpose is for visualization the best practice combination algorithm for keypoint detection is SURF as it results in more reconstructed points than SIFT, however if the purpose is for camera tracking then the best practice combination algorithm for keypoint detection is SIFT. For all purposes the best practice combination algorithm for matching is optical flow as it is the most efficient and the best practice combination algorithm for ego-motion estimation is the 5-point algorithm as it is robust to points located on planes. Therefore the best practice combinations are *SURF - OpticalFlow - 5-point* and *SIFT - OpticalFlow - 5-point.*

A multi view dataset called *Fountain-P11* was reconstructed (Strecha et al., 2008). The camera positions determined using the best practice combination *SURF - OpticalFlow - 5-point* had an average error of 0.119m with a standard deviation of 0.093m and resulted in 22029 points being reconstructed in 867 seconds. The best practice combination *SIFT*

- *OpticalFlow - 5-point* had an average camera position error of 0.114m with a standard deviation of 0.095m and resulted in 3113 points being reconstructed in 187 seconds.

The main implication of this research is that progressive stereo reconstruction can be performed in near real-time on a mobile device without compromising the accuracy of reconstruction.

This research is significant as the effects of the key steps of progressive reconstruction and the choices made at those steps on the accuracy and efficiency of the reconstruction as a whole have never been studied before.

It is recommended that the above findings are tested on more datasets, in particular indoor datasets and that the scalability of the algorithm is tested i.e the algorithm is tested for more than 10 frames.

Technically it is recommended that:

1. The accuracy of the fundamental matrix is considered. The accuracy of the fundamental matrix can be computed by determining the covariance matrices.
2. The more efficient 8-point algorithm is used and a check is put in place to ensure the selection of points used to compute it are not degenerate. To check for an ill-conditioned fundamental matrix (i.e a degenerate configuration) the dimensions of the A matrix can be observed. If the matrix is degenerate the dimensions will be equal to two or three, if the matrix is not the dimensions will be equal to one.
3. If the points are noisy a least squares approach is used to compute the fundamental matrix.
4. If there is a lack of texture in the scene then descriptor based matching should be performed instead of optical flow based matching as optical flow relies on texture.
5. The accuracy and efficiency of the algorithm is compared to itself with a best practice approach to reduce drift i.e the Kalman filter and bundle adjustment.

The algorithm is restricted to side-ways camera motion as ego-motion estimation in the forward direction is highly inaccurate. Further research directions could be in the field of progressive stereo reconstruction in forward motion.

Appendix A

Background To Digital Image Processing

This chapter presents a background to digital image processing and relevant best practice algorithms. There are six sections , namely: intensity surfaces, linear filters,image transformations,perspective distortion ,motion flow and optical flow

This chapter was synthesized from lecture notes produced by Robert Collins for the computer vision CSE/EE486 course at the Department of Computer Science and Engineering at Penn State University (Owens, 1997).

The section on perspective distortion was synthesized from an assignment produced by Chad Aeschliman for the computer vision ECE661 course at the College of Engineering at Purdue University (Aeschliman, 2008).

A.1 Intensity Surfaces

An image is essentially a surface of pixel values ranging from 0-255 (black-white) i.e an intensity surface.

A.1.1 Gradient

An image does not have an algebraic function , therefore to work out the gradient of the surface at a point numerical derivatives are used. A numerical derivative is calculated by a linear combination of the values of neighboring pixels. The derivative with respect to x/y is a combination of neighboring pixels in the x/y direction .

$$I_x = \frac{dI(x,y)}{dx} = \frac{I(x+1,y) - I(x-1,y)}{2} \tag{A.1}$$

$$I_y = \frac{dI(x,y)}{dy} = \frac{I(x,y+1) - I(x,y-1)}{2} \tag{A.2}$$

A.1.2 Gradient Magnitude

The resulting pixel values from the derivative in the x and y direction can be combined to create a magnitude of gradient image.

$$Mag = \sqrt{I_x^2 + I_y^2} \tag{A.3}$$

A.1.3 Gradient Angle

The resulting pixel values can also be used to create an angle of gradient image. The angle of gradient is great for visualizing detail in low contrast areas, such as indoor spaces.

$$Angle = a\tan 2(\frac{I_y}{I_x}) \tag{A.4}$$

A.2 Linear Filters

Linear filtering is a process whereby a new image is formed whose pixels are a weighted linear combination of neighboring pixels.

The linear filter takes the form of a matrix of weights referred to as the kernel. The kernel is convolved with the image. The kernel must be rotated 180 degrees prior to convolving otherwise cross correlation should be used.

When the kernel is convolving the pixels towards the outside it may extend outside of the image. As a result the overlapping areas are normally treated as zero pixels as seen below in A.1, however there are different ways of treating the overlapping areas such as replication, reflection and wraparound.

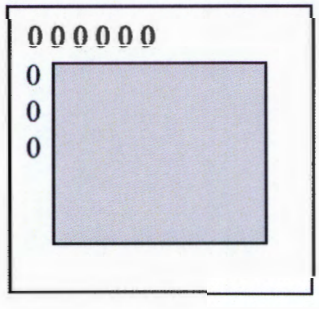


FIGURE A.1: Zero padding around an image

A.2.1 Derivative

The derivative filter is either in the x or y direction. The derivative filter in the x directions convolution kernel is $\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}^T$ in the y direction.

A.2.2 Smoothing

Camera sensors receive light and process it into electrical charges whose outputs are pixels in the the image. The electrical chargers tell the sensor what color each pixel is meant to be.

The amount of noise depends on the capturing conditions and the properties of the camera, for example images containing shadows will have higher sensor noise in those areas and cameras with a larger resolution and small sensor size will have more tightly

packed pixels on the sensor resulting in an higher chance that overheating electrons will corrupt the light sensors.

Smoothing is used to remove sensor noise. There are two types of smoothing filters: a Box filter and a Gaussian filter. The Box filter replaces each pixel with an equally weighted average of its neighboring pixel values whereas the Gaussian replaces each pixel with a Gaussian weighted average of its neighboring pixel values.

Smoothing should be the first filter applied to an image before any other operations are performed. Smoothing reduces the amount of detail in an image and therefore not all pixels need to be preserved as some are redundant. It is better to slowly reduce the number of pixels as one smoothes more and more. The Gaussian filter makes incremental smoothing possible. Therefore after every smoothing operation the image should be down sampled i.e. pixels should be removed and thus the image reduced in size. Thereafter the image should be up sampled and the empty pixels filled by interpolation (the empty pixels are set to zero and the image pixels are convolved with a Gaussian filter). It is crucial to smooth before down sampling to avoid aliasing.

It is possible to unsmooth an image i.e. to add noise. The noise may be Gaussian, Multiplicative or Impulse (Salt and paper).

A.2.3 Smoothing + Derivative

There are two filters which combine a smoothing filter and derivative filter into a single filter: the Prewitt filter and Sobel filter. The Prewitt filter involves smoothing with a Box filter and the Sobel filter with a Gaussian filter. These filters are furthermore horizontal/vertical depending on whether the derivative chosen is in the x/y direction.

A.2.4 Smoothed Derivative

A single filter exists that is inherently a smoother, namely the Derivative of Gaussian.

If a filter is repeatedly convolved with itself it takes on a Gaussian shape, the standard deviation of the resulting shape is equal to the sum of the standard deviations of repeated convolutions. The standard deviation of the resulting convolution achieved by reapplying the convolution filter [1 1] can be determined by Pascal's triangle.

A.2.5 Fourier Transform

The Fourier transform is a mathematical transformation to transform an image/function from the spatial domain to the frequency domain. It is reversible. The Fourier transform involves taking a function and decomposing it into sin and cosine components (this process is reversible) in the frequency domain. After applying the fourier transform the resulting function is an energy function. Plotting this function for various frequencies illustrates where the energy is concentrated and how quickly it dies off.

It is often easier to perform certain operations such as filtering in the frequency domain than in the spatial domain. For example, smoothing is used to remove noise (high frequency information) in an image i.e it is a low pass filter. A low pass filter can easily be applied in the frequency domain. As seen in figure A.2 an image is transformed into the frequency domain by applying a Fourier transform to it. Most of the image information is located in the center (low frequencies) of the Fourier transform image. To apply a low pass filter image information away from this center is deleted and then the Fourier domain is transformed back into the spatial domain and a smoothed image is presented that is significantly reduced in size thanks to the removal of noise.

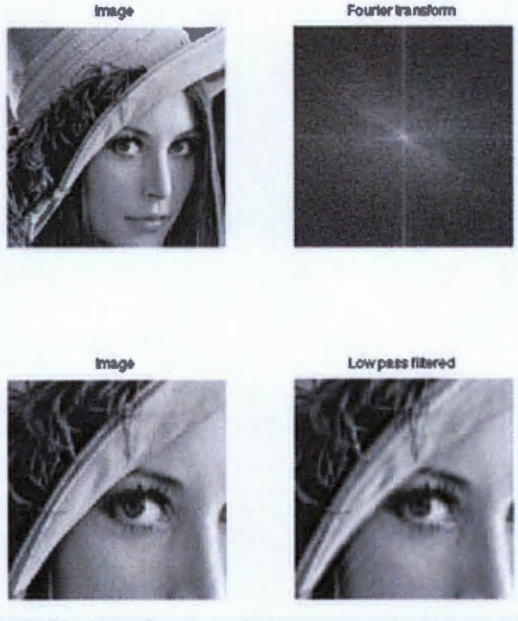


FIGURE A.2: Smoothing applied to an image in the fourier domain (Peyré, 2012)

A.3 Image Transformations

A.3.1 Types Of Transformations

There are four types of 2D transformations, in order of preserving less: euclidean, similarity , affine and projective.

A.3.1.1 Euclidean

A euclidean transformation preserves lengths and angles. The euclidean transforms are rotation and translation. The transformation has three degrees of freedom (Translation (2), Rotation (1)).

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.5}$$

A.3.1.2 Similarity/Metric

A similarity/metric transformation preserves shape i.e angles between lines and ratios of lengths. The similarity/metric transforms are rotation, translation and scaling. The transformation has four degrees of freedom (Scale (1),Translation (2), Rotation (1)).

$$\begin{bmatrix} sr_{11} & sr_{12} & t_x \\ sr_{21} & sr_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.6}$$

A.3.1.3 Affine

A affine transformation preserves ratios of areas , parallel lines and planes. It is a linear transformation followed by a translation (can be represented by 1 single 3x3 matrix by homogeneous notation). The affine transforms are rotation, translation,scaling and shear. The transformation has six degrees of freedom (a(4),Translation (2)).

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.7}$$

A.3.1.4 Projective

A projective transformation preserves collinearity. It is a linear transformation of homogeneous n vectors represented by a non-singular nxn matrix. The projective transforms are rotation, translation,scaling and shear. The transformation has eight degrees of freedom (h(8)).

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.8}$$

A.3.2 Warping

Warping is the application of a transformation on an image (e.g affine/ projective). There are two types of warping: forward and inverse.

A.3.2.1 Forward Warping

The image pixels from the image to be warped are sent to their corresponding location in the new grid. If the pixel lands between two pixels in the new grid the color of the pixel is distributed among neighboring pixels in the new grid ,for example by splatting . Forward warping can be seen in figure A.3.

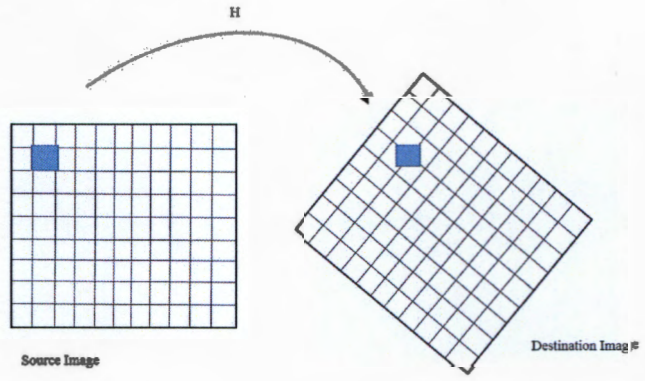


FIGURE A.3: Forward warping. The image pixel is sent to its new location in the new grid, if it lands between two pixels in the new grid its color will be distributed amongst the neighbors

A.3.2.2 Inverse Warping

Each image pixel in the new grid is replaced with the pixel from its corresponding location in the original image. If the pixel in the new grid comes from between two pixels in the image the color value is interpolated from its neighbors. Examples of interpolation methods are nearest neighbor, average and Gaussian average. Inverse warping is preferred as it eliminates the possibility of holes. Inverse warping can be seen in figure A.4.

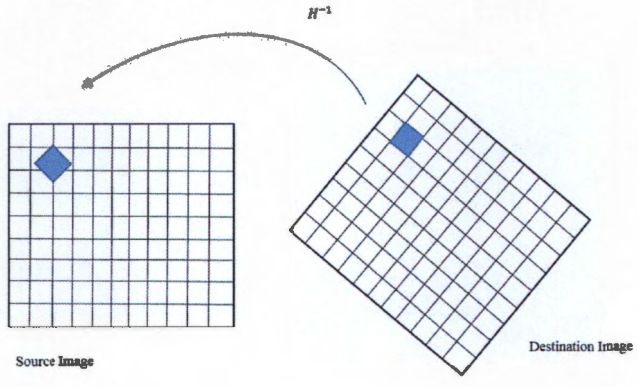


FIGURE A.4: Inverse warping. Each pixel in the new image is replaced from the pixel at its corresponding location. If it lies between two pixels it is interpolated from its neighbors

A.4 Perspective Distortion

Perspective is one of the ways in which the human eye judges depth. It refers to the angle and location of parallel lines in the scene, when parallel lines start converging the eye registers depth. For example when one is looking down a road the road will appear to narrow the further it is from you. Vertical lines and edges will always remain vertical to the human eye. Cameras over the years have replicated the human eye so to make images look as real as possible, this is known as the perspective projection . A orthographic (parallel) projection on the other hand preserves parallel lines.

Perspective distortion is a type of distortion where vertical lines in an image begin to converge. It is caused by the cameras image plane not being parallel to the object plane or not level with the center of an object.

Homographies can be used to bring the image planes parallel and in level with the center of the object i.e change the perspective of the image and remove perspective distortion.

Homography is a term that describes the relationship between images and a planar surface in the scene and the relationship between images that are captured by a rotating camera. These two relationships can be seen in figure A.5. The matrix which quantifies this relationship is known as a homographic matrix, it normally is a projective transformation.

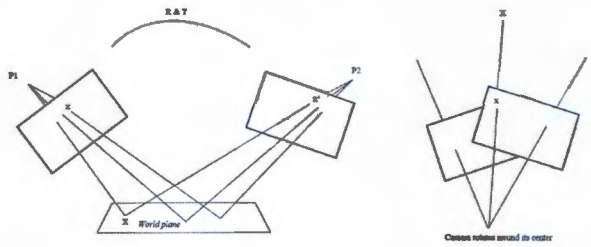


FIGURE A.5: Two types of homography: Images of a planar surface and images captured by a rotating camera

A homography matrix can therefore be built that brings the plane in the scene parallel to the camera. The homography matrix is built using two sets of points. The first set of points is fictitious and accurately represents the true dimensions of a planar surface in the scene. The second set is the corresponding image points.

It is common to use the point matches to build a projective homography matrix. A projective transformation only preserves collinearity and removes the most perspective distortion.

If it is decided to build a similarity homography matrix in order to remove as much perspective distortion as possible whilst preserving the ratio of lengths in the original image then a projective homography matrix is applied to the image and the projective correction and affine corrections are removed.

A.5 Motion Flow

The motion of a camera can be projected onto the image of the scene as 2D velocity vectors at each pixel. The 2D projection of the motion of the camera between two frames can be seen in figure A.6.

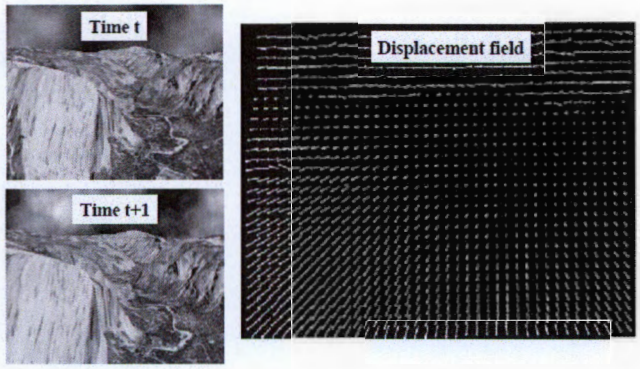


FIGURE A.6: Motion field showing the projection of 3D relative velocity vectors onto the image (Owens, 1997).

In order to create the motion field the following steps are followed :

1. Determine the displacement of the camera between frames.

   The displacement is equal to (RP+T)-P where R and T are the rotation and translation between frames and P is the position of the camera at the first frame.

2. Write the rotation matrix in euler angles: θ ϕ and ψ

   Euler angles are three angles ψ , θ, ϕ that describe the orientation of an object . They represent a sequence of rotations about an axes of a coordinate system. In the below equations c represents cos and s represents sin.

   The euler rotation angle for rotation around the z axis (pitch ψ):

$$R_z = \begin{bmatrix} c(\psi) & -s(\psi) & 0 \\ s(\psi) & c(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.9}$$

   The euler rotation angle for rotation around the y axis (yaw θ):

$$R_y = \begin{bmatrix} \mathrm{c}(\theta) & 0 & \mathrm{s}(\theta) \\ 0 & 1 & 0 \\ -\mathrm{s}(\theta) & 0 & \mathrm{c}(\theta) \end{bmatrix} \tag{A.10}$$

The euler rotation angle for rotation around the x axis (roll ϕ):

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}(\phi) & -\mathrm{s}(\phi) \\ 0 & -\mathrm{s}(\phi) & \mathrm{c}(\phi) \end{bmatrix} \tag{A.11}$$

The sequence in which they are applied is crucial to avoid gimbal lock. Gimbal lock occurs when rotation among two axis will result in the same rotation.

The order of rotation chosen is:

$$R = R_Z R_y R_x \tag{A.12}$$

Therefore the euler rotation matrix is:

$$R = \begin{bmatrix} \mathrm{c}(\psi)\mathrm{c}(\theta) & -\mathrm{s}(\psi)\mathrm{c}(\phi) + \mathrm{c}(\psi)\mathrm{s}(\theta)\mathrm{s}(\phi) & \mathrm{s}(\psi)\mathrm{s}(\phi) + \mathrm{c}(\psi)\mathrm{c}(\phi)\mathrm{s}(\theta) \\ \mathrm{s}(\psi)\mathrm{c}(\theta) & \mathrm{c}(\psi)\mathrm{c}(\phi) + \mathrm{s}(\phi)\mathrm{s}(\theta)\mathrm{s}(\psi) & -\mathrm{c}(\psi)\mathrm{s}(\phi) + \mathrm{s}(\theta)\mathrm{s}(\psi)\mathrm{c}(\phi) \\ -\mathrm{s}(\theta) & \mathrm{c}(\theta)\mathrm{s}(\phi) & \mathrm{c}(\theta)\mathrm{c}(\phi) \end{bmatrix} \tag{A.13}$$

For small angles cosx=1 and sinx=x so the rotation matrix can be simplified to:

$$R = 1 + \begin{bmatrix} 0 & -\psi & \theta \\ \psi & 0 & -\phi \\ -\theta & \phi & 0 \end{bmatrix} = 1 + S \tag{A.14}$$

Therefore displacement becomes (1+S)P+T-P = SP+T = T+SP

$$SP = [\theta_x, \theta_y, \theta_z]XP = \omega XP \tag{A.15}$$

3. Determine the 3D velocity between frames.

   As the time between frames is less than 1/30 seconds the 3D displacement is equal to 3D velocity (linear velocity + angular velocity)

$$V = T + \omega XP \tag{A.16}$$

4. Determine the projection of this 3D velocity on the image.

   If the cameras axis are in line with the worlds axis then the world coordinates are the same as the camera coordinates. Therefore the perspective projection geometry equation relates the world coordinates to coordinates in the image plane.

$$p = \frac{fP}{Z} \tag{A.17}$$

   The derivative of the perspective projection equation is taken and manipulated to give the following:

$$v = f\frac{V}{Z} - p\frac{V_z}{Z} \tag{A.18}$$

   substituting in the 3D velocity the following motion flow equations are found

$$v_x = \frac{T_z x - T_x f}{Z} - \omega_y f + \omega_z y + \frac{\omega_x xy}{f} - \frac{\omega_y x^2}{f} \tag{A.19}$$

$$v_y = \frac{T_z y - T_y f}{Z} + \omega_x f - \omega_z x + \frac{\omega_y xy}{f} + \frac{\omega_x y^2}{f} \tag{A.20}$$

The first component of the motion flow equation is the translational component and the remaining components are the rotational components which are independent of the depth of the scene (Z value). This can be seen in image A.7 below. As the translational component varies with the depth of the scene it exhibits motion parallax (objects closer appear to move faster across the field of view). Therefore it is often useful to remove translational motion in order to determine the steering angles ωx and ωy (given ωz = 0) of a car that is driving into the scene , for example. The rotational component can be seen in A.7. It can be removed by dot-producting the flow with a new vector that will make it zero. Then the ωx and ωy that solve n linear equations can be found.

In the special case that the camera only translates and does not rotate between frames then the 2D velocity will be radial (if the camera is moving towards the scene the rays will point outwards and vice versa). If however the camera is moving parallel to the scene the motion flow vectors will be parallel to each other. If the camera captures a planar scene, then the Z value of the scene is constant and less parameters are needed to solve for the 2D velocity vectors.

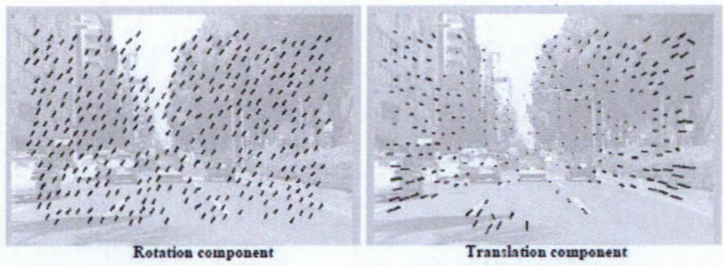


FIGURE A.7: Decomposition of vehicle flow field into rotational and translational components (Owens, 1997).

A.6 Optical Flow

Optical flow is the apparent motion of image objects between two frames caused by movement of the object or camera. It can be used to approximate motion flow when the rotation and translation of the camera between frames is unknown. An example of the optical flow in an image can be seen in figure A.8.

FIGURE A.8: Optical flow. The apparent motion of the camera/object between frames (Owens, 1997).

Optical flow is based on two assumptions:

1. The intensity of pixels do not change between consecutive frames. This is known as the brightness constancy constraint

$$I(x(t), y(t), t) = Constant \tag{A.21}$$

2. Neighboring pixels have similar motion

The derivative of the brightness constancy equation is:

$$\frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial I}{\partial t} = 0 \tag{A.22}$$

$$I_x u + I_y v + I_t = 0 \tag{A.23}$$

The above equation is called the optical flow equation. I_x and I_y are the spatial gradients, I_t is the temporal gradient and u and v are the flow vector components. Optical flow

is therefore constrained to be in the direction of the spatial image gradient i.e normal flow. The component of flow parallel to an edge is unknown. There are several methods for solving the equation for the flow vector components. The most common being the differential methods.

Some areas of optical flow are not in the direction of true motion . This is known as the aperture problem and is common at edges.

A.6.0.3 Lucas-Kanade Method

The Lucas Kanade method is based on the assumption that the flow (u,v) is constant for all pixels in a 5x5 patch.

$$\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ .. & .. \\ I_x(p_{25}) & I_y(p_{25}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ .. \\ I_t(p_{25}) \end{bmatrix} \tag{A.24}$$

There are more equations than unknowns so a least squares solution is required:

$$Ad = b \tag{A.25}$$

$$A^T Ad = A^T b \tag{A.26}$$

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{A.27}$$

For the above equation to be solvable $A^T A$ must be invertible. To avoid noise its eigenvalue λ_2 should not be too small. If λ_1 / λ_2 is too large it means the point is on an edge. Edges often suffer from the aperture problem. Therefore to work properly λ_1 and λ_2 must be large enough and have a similar magnitude. This condition is similar for the Harris corner detection and therefore it is better to use the Lucas Kanade method only at corner points

Appendix B

Background To Single View Geometry

This chapter presents a background to single view geometry and relevant best practice algorithms. There are two sections , namely: the pinhole camera model and camera calibration. The camera calibration section introduces the absolute conic, methods of calibrating the camera and finishes off giving an overview of lens distortion.

This chapter was synthesized from chapter two *Camera Calibration* of the book *Emerging Topics in Computer Vision* by Gerard Medioni and Sing Kang (Medioni and Kang, 2004)
.

B.1 The Pinhole Camera Model

The pinhole camera model relates the world,camera and image coordinate systems. It assumes the optical center of the camera, the image point and the world point are collinear. The model can be seen in figure B.1.

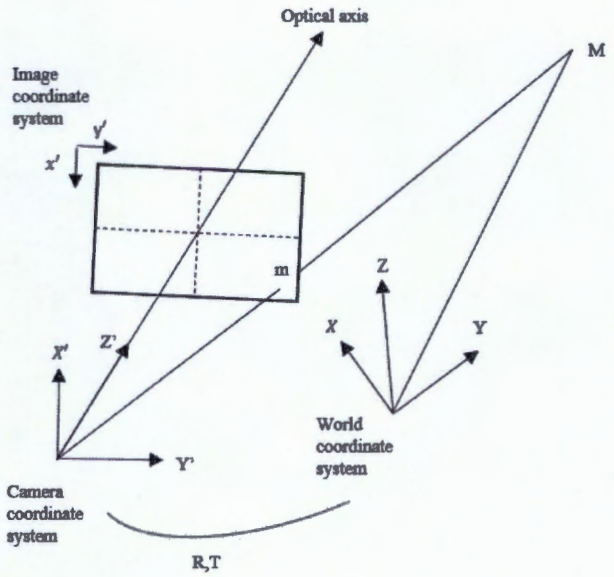


FIGURE B.1: The pinhole camera model. It relates the world, camera and image coordinate systems.

In homogeneous coordinates a image point is denoted by m= $[u, v, 1]^T$ and a 3D point is denoted by M= $[X, Y, Z, 1]^T$. The relationship between m and M is represented by the camera matrix P.

$$m = PM \tag{B.1}$$

The camera matrix P mixes both intrinsic (K) and extrinsic parameters(R,T). This can be seen below in the below projective equation:

$$m = K \begin{bmatrix} R & T \end{bmatrix} M \tag{B.2}$$

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{B.3}$$

(R,T) are the extrinsic parameters of rotation and translation that relate the camera coordinate system to the world coordinate system. The extrinsic parameter R represents the rotation of the world coordinate system w.r.t the camera coordinate system and the extrinsic parameter T represents the location of the origin of the world coordinate system in the camera coordinate system.

K is the camera intrinsic matrix with (c_x, c_y) the coordinates of the principal point, f_x and f_x are the focal length in x and y units (i.e the scale factors in x and y direction), and γ is the parameter describing the skew of the two image axes. This matrix is known as the calibration matrix and is required in order to perform a similarity/metric reconstruction of the scene from two images. θ is the angle between the image axes, if the pixels are rectangular then $\theta = 90$ and $\gamma = 0$.

B.2 Camera Calibration

Camera calibration is used to determine the intrinsic and extrinsic parameters of the camera. The intrinsic parameters are enclosed in what is known as the calibration matrix. These intrinsic parameters are required in order to perform a metric/similarity reconstruction.

B.2.1 The Absolute Conic

In order to determine the cameras intrinsic parameters the image of the absolute conic Ω needs to be determined.

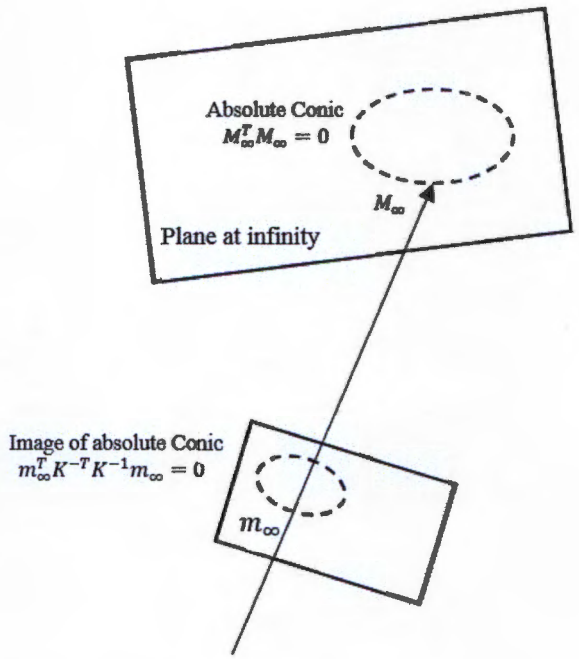


FIGURE B.2: The absolute conic and its image

Points M_∞ which are on the plane at infinity (x_4=0) have homogeneous coordinates

$$\tilde{M}_\infty = \begin{bmatrix} M_\infty \\ 0 \end{bmatrix} \tag{B.4}$$

The absolute conic Ω is defined by a selection of points on the plane at infinity which satisfy the following condition:

$$M_1^2 + M_2^2 + M_3^2 = 0 \tag{B.5}$$

Points on the absolute conic have the following two properties:

1. They satisfy the equation:
$$M^T{}_\infty M_\infty = 0 \tag{B.6}$$

2. They are invariant to rigid transformations H.
$$H = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{B.7}$$
$$\tilde{M}'_\infty = H\tilde{M}_\infty = \begin{bmatrix} Rm_\infty \\ 0 \end{bmatrix} \tag{B.8}$$
The transformed points are still on the plane at infinity and are also on the same conic Ω.
$$M'^T{}_\infty M'_\infty = (RM_\infty)^T(RM_\infty) = M_\infty^T(R^T R)M_\infty = 0 \tag{B.9}$$

As the points on the absolute conic are invariant to transformations the relative position of the absolute conic to a moving camera is constant. If the intrinsic parameters are constant the image of the absolute conic will also be constant.

The projection of a point on the absolute conic to the image is:

$$\tilde{m}_\infty = K[\ R \quad T\ ] \begin{bmatrix} x_\infty \\ 0 \end{bmatrix} \tag{B.10}$$

The projected point also satisfies the first property of a conic.

$$\tilde{m}_\infty^T K^{-T} K^{-1} \tilde{m}_\infty = 0 \tag{B.11}$$

$$\omega = K^{-T} K^{-1} \tag{B.12}$$

Therefore the image of the absolute conic is also a conic ω and is determined by the intrinsic parameters of the camera i.e the calibration matrix K. If the image of the

absolute conic ω is found the cameras intrinsic parameters can be determined. There are various camera calibration techniques with different approaches to finding the image of the absolute conic.

B.2.2 Calibrating The Camera Using A 3D Object

This is the traditional way to calibrate a camera and is the most accurate. A random object such as a cube is selected and a 3D coordinate system is attached to this object, a checkerboard pattern is displayed on the object and the coordinates of the corners are known very accurately. This method first determines the camera matrix and then decomposes it into the intrinsic and extrinsic parameters.

B.2.2.1 Determining The Camera Matrix

The first step is to take a sequence of images of the object from different positions and detect the corners of the checkerboard pattern using a corner detector such as the Harris corner detector. A more accurate solution is to detect edges and then fit a line to the edges and then compute the corners by the intersection of the lines. Once the corner points in the image $m_i = (u_i, v_i)$ are known their corresponding points in 3D space $M_i = (X_i, Y_i, Z_i)$ can easily be found as the pattern is known.

To estimate the camera matrix P the following linear set of equations Gp=0 are formed:

$$\begin{bmatrix} X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & u_iX_i & u_iY_i & u_iZ_i & u_i \\ 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & v_iX_i & v_iY_i & v_iZ_i & v_i \end{bmatrix} p = 0 \tag{B.13}$$

where $p = [p_{11}, p_{12}, ..., p_{34}]^T$ and $0 = [0, 0]^T$

For n point matches matrix G will become a 2nx12 matrix. The least squares solution for p is the singular vector which corresponds to the smallest singular value of G. This vector p is the last column of V in the SVD $G = UDV^T$. It minimizes $||Gp||$ subject to the condition $||p||$=1. The vector p makes up P. If the point coordinates are noisy the solution will be biased

B.2.2.2 Determining The Intrinsic And Extrinsic Parameters From The Camera Matrix

Once the camera matrix P is known the intrinsic and extrinsic parameters can be computed as follows:

$$P = K \begin{bmatrix} R & t \end{bmatrix} \tag{B.14}$$

$$B = KR \tag{B.15}$$

$$b = Kt \tag{B.16}$$

$$J = BB^T = KK^T = \begin{bmatrix} \kappa_u & \kappa_c & c_x \\ \kappa_c & \kappa_v & c_y \\ c_x & c_y & 1 \end{bmatrix} \tag{B.17}$$

k_{33} may not be equal to 1 and therefore J might have to be normalized to ensure it equals 1.

The intrinsic parameters are therefore:

$$c_x = J_{13} \tag{B.18}$$

$$c_y = J_{23} \tag{B.19}$$

$$f_y = \sqrt{\kappa_v - c_y^2} \tag{B.20}$$

$$\gamma = \frac{\kappa_c - c_x c_y}{f_y} \tag{B.21}$$

$$f_x = \sqrt{\kappa_u - c_x^2 - \gamma^2} \tag{B.22}$$

The extrinsic parameters are therefore:

$$R = K^{-1}B \tag{B.23}$$

$$T = K^{-1}b \tag{B.24}$$

The intrinsic and extrinsic parameters are affected by noise in the point coordinates. A more accurate version of the parameters can be obtained by the the maximum likelihood estimate. The maximum likelihood estimate is obtained by minimizing the distances between the image points $m_i = (u_i, v_i)$ and their reprojected positions i.e the re-projection error. This is a non-linear minimization problem which can be solved using the Levenberg-Marquardt Algorithm. The parameters worked out linearly above can be used as provisionals.

$$\sum_i \|m_i - \phi(P, M_i)\|^2 \tag{B.25}$$

The Levenberg-Marquardt Algorithm is an iterative procedure. The algorithm can be explained by thinking of minimizing a function as finding the lowest point on a surface, in order to find such a point the surface is traversed in a downhill direction until the surface no longer goes downhill,then a new direction that goes downhill is found and followed , this is repeated until there is a point where no direction goes downhill anymore.

Suppose the function F is being minimized and this function represents the distance between image points and their reprojection positions. The first iteration is at the point x(n), it is desired that the next iteration is at the point x(n+1) such that f(x(n+1))<f(x(n)) i.e the function value/distance is smaller. The Levenberg-Marquardt Algorithm achieved this by determining the linear approximation value at the current iteration x(n) and then finding the downhill direction, if the linear approximation is a good approximation of F at x(n) a large step in that downhill direction can be taken, if not a small step is taken. This is repeated.

B.2.3 Calibrating The Camera Using A Moving Plane

The most convenient way of determining the calibration matrix is the plane-based technique , this involves simply taking images of a moving plane that has a 2D pattern e.g. a checkerboard. Since all the points lie on a plane their z scene coordinate is 0 .

The projective equation then reduces to:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{B.26}$$

The real world plane and its image are then related by a homography H:

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix} \tag{B.27}$$

$$(h_1, h_2, h_3) = K\,(r_1, r_2, t) \tag{B.28}$$

$$r_1 = K^{-1} h_1 \tag{B.29}$$

$$r_2 = K^{-1} h_2 \tag{B.30}$$

As r_1 and r_2 are orthonormal the following two conditions/equations exist:

$$h_1^T K^{-T} K^{-1} h_2 = 0 \tag{B.31}$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2^T K^{-T} K^{-1} h_2 = 0 \tag{B.32}$$

Therefore each plane provides the above two equations.

$$B = K^{-T}K^{-1} = \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{pmatrix} \tag{B.33}$$

$$b = (B_{11}, B_{12}, B_{13}, B_{22}, B_{23}, B_{33}) \tag{B.34}$$

B has 6 degrees of freedom therefore as each plane provides two equations 3 different views of the plane are needed to solve for B. Each plane must have 4 points.

Using the two conditions and b the system of final equations: Vb=0 is developed and solved by least squares.

The intrinsic parameters are therefore:

$$C_y = \frac{(B_{12}B_{13} - B_{11}B_{23})}{(B_{11}B_{22} - B_{12}^2)} \tag{B.35}$$

$$\lambda = B_{33} - \frac{B_{13}^2 + C_y(B_{12}B_{13} - B_{11}B_{23})}{B_{11}} \tag{B.36}$$

$$f_x = \sqrt{\frac{\lambda}{B_{11}}} \tag{B.37}$$

$$f_y = \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}} \tag{B.38}$$

$$\gamma = -\frac{B_{22}{f_x}^2 f_y}{\lambda} \tag{B.39}$$

$$C_x = \frac{\gamma C_y}{\alpha} - \frac{B_{13}{f_x}^2}{\lambda} \tag{B.40}$$

The extrinsic parameters are therefore:

$$r_1 = \lambda K^{-1} h_1 \tag{B.41}$$

$$r_2 = \lambda K^{-1} h_2 \tag{B.42}$$

$$r_3 = r_1 x r_2 \tag{B.43}$$

$$T = \lambda K^{-1} h_3 \tag{B.44}$$

$$R = [r_1, r_2, r_3] \tag{B.45}$$

R is parameterized by a vector r of 3 parameters , it is parallel to the rotation axis and its magnitude is equal to the rotation angle. R are r are related by the Rodrigues formula.

The intrinsic and extrinsic parameters are affected by noise in the point coordinates. A more accurate version of the parameters can be obtained by the the maximum likelihood estimate. The maximum likelihood estimate is obtained by minimizing the distances between the image points $m_i = (u_i, v_i)$ and their reprojected positions i.e the re-projection error. This is a non-linear minimization problem which can be solved with the Levenberg-Marquardt Algorithm. The parameters worked out linearly above can be used as provisionals.

$$\sum_{i=1}^{n} \sum_{j=1}^{m} \| m_{ij} - \hat{m}(K, R_i, t_i, M_j) \|^2 \tag{B.46}$$

B.2.4 Alternative Methods Of Camera Calibration

It is possible to determine the calibration matrix by observing a set of collinear points such a string of balls hanging from the ceiling.

It is possible to determine the calibration matrix by not observing any calibration object i.e a 0D approach (self-calibration). In this approach all that is required are image correspondences between three images. This approach is less accurate than when a calibration object is used as a large number of parameters need to be estimated which means a much harder mathematical problem.

B.2.5 Lens Distortion

Pinhole cameras do not have lenses therefore according to the pinhole model the 3D point, its corresponding image point and the optical center of the camera are collinear.

For some modern day cameras this is not always true. Most modern day cameras have lenses and low-end/wide-angle cameras have a sufficient amount of lens distortion which prevents this collinearity.

There are two types of lens distortion: radial and tangential (decentering) distortion. Radial distortion is symmetric and is caused by an imperfect lens shape. Tangential distortion is caused by an improper lens assembly. Both distortions increase with distance r from the distortion center.

The image coordinates can be corrected for distortion by distortion corrections δ_x and δ_y:

$$\breve{x} = x + \delta_x \tag{B.47}$$

$$\breve{y} = y + \delta_y \tag{B.48}$$

$$\delta_x = x(k_1 r^2 + k_2 r^4 + k_3 r^6 + ...) + [p_1(r^2 + 2x^2) + 2p_2 xy](1 + p_3 r^2 + ...) \tag{B.49}$$

$$\delta_y = y(k_1 r^2 + k_2 r^4 + k_3 r^6 + ...) + [2p_1 xy + p_2(r^2 + 2y^2)](1 + p_3 r^2 + ...) \tag{B.50}$$

k_i and p_i are the coefficients of radial and tangential distortion. The majority of the distortion is radial and therefore generally the distortion is equal to only the first radial distortion terms κ_1 and κ_2

After the calibration matrix and other parameters have been determined the lens distortions κ_1 and κ_2 can be determined by minimizing the below non-linear function using the Levenberg-Marquardt method. The provisional values for κ_1 and κ_2 are zero.

$$\min(K, R, t, \kappa_1, \kappa_2, M_i) = \sum_i \|m_i - \phi(K, R, t, \kappa_1, \kappa_2, M_i)\|^2 \tag{B.51}$$

Appendix C

Background To Two-View Geometry

This chapter presents a background to two-view geometry (the geometry from two views) and relevant best practice algorithms. There are four sections , namely: keypoint detection and matching, epipolar geometry, the camera matrices and triangulation.

This chapter was synthesized from the book *Multiple view geometry in computer vision* by Richard Hartley and Andrew Zisserman (Hartley and Zisserman, 2003) . This book is universally accepted as the "bible" of Multiple View Geometry.

The first section was synthesized from the lecture notes on stereo matching produced by Robyn Owens for the computer vision IT412 course at the School of Informatics at University of Edingburgh (Owens, 1997) and the lecture notes produced by Robert Collins for the Computer Vision CSE/EE486 course at the Department of Computer Science and Engineering at Penn State University (Collins, 2007).

The section on determining the essential matrix from 5 point correspondences was synthesized from the paper (Nistér, 2004).

C.1 Keypoint Detection And Matching

The first step in 3D reconstruction is to determine point correspondences between overlapping images. It may be difficult to find correspondences because some points in each image will have no corresponding points in the other image, either because the cameras have different fields of views or due to occlusion. The purpose of finding point correspondences is to create a fundamental matrix which will be discussed in the next section. One point should always only match one point in the corresponding image. To ensure accurate matches the correspondences can be pruned using RANSAC which will be described later.

There are two approaches to determining point correspondences, namely intensity-based matching and feature-based matching.

C.1.1 Intensity-Based Matching

Intensity-based matching compares the intensity values of pixels in patches. It is computationally exhaustive to compare patches at all points and therefore candidate point correspondences are chosen in interesting regions i.e regions that have a high variation of intensity values in the horizontal, vertical and diagonal directions.

To find such points each image can be convolved with the common Moravec filter . It calculates the minimum mean square difference in intensity values for a window shifted in the four main directions. If the value is above a threshold the point is marked as interesting. The Moravec filter is easy to implement, light on computation however its not rotationally invariant.

The size of the patch is important, if the patch is too small the patch will not capture enough intensity variation (and therefore not be distinct enough) and may be sensitive to noise resulting in false matches. If the patch is too big the matching will be less sensitive to noise but also to actual variations in intensity.

The most serious shortcoming of this approach is its sensitivity to foreshortening. The effect of foreshortening can be seen in figure C.1 where the size of the image projection of the scene captured changes with viewing position and direction . The best correlation would be achieved if the patch size in each image was dependent on the size of the image projection, however this is not possible without knowledge of the scene.

The success of this approach depends on whether the image patch in one image exhibits a distinctive structure that occurs infrequently in the search space of the other image.

To improve computational efficiency of matching the images can be broken up into a pair of image pyramids (hierarchy of images) and the matching process can be started at the coarser level (lowest resolution), and then the matches obtained can be used to guide the matching up until the highest level is reached.

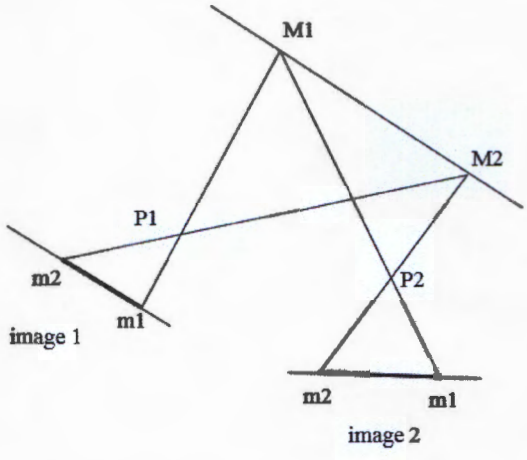


FIGURE C.1: Foreshortening: The size of the image projection of the scene captured changes with viewing position and direction

C.1.1.1 Correlation Measures

There a number of correlation measures which can be used to find the best matches, such as cross-correlation , sum of squared distances (SSD), max difference, sum of absolute differences (SAD) and normalized cross-correlation (NCC)

C.1.1.1.1 Cross-Correlation

Cross correlation involves taking the product of each pixel in the patches and then summing them together. The point with the high similarity score i.e sum is the corresponding point.

$$C_{fg} = \sum_{[i,j] \in R} f(i,j)g(i,j) \tag{C.1}$$

C.1.1.1.2 Sum Of Squared Distances (SSD)

Sum of absolute difference involves taking the squared difference between each pixel in the patches and then summing them together. The point with the lowest similarity score i.e sum is the corresponding point. Sum of squared distances is the most popular correlation measure and is more accurate than cross-correlation.

$$SSD = \sum_{[i,j] \in R} (f(i,j) - g(i,j))^2 \tag{C.2}$$

C.1.1.1.3 Max Difference

Max difference involves taking the difference between each pixel in the patches and then looking for the largest difference. The point with the lowest similarity score (i.e lowest largest difference) is the corresponding point.

$$max[i,j] \in |f(i,j) - g(i,j)| \tag{C.3}$$

C.1.1.1.4 Sum Of Absolute Differences (SAD)

Sum of absolute difference simply involves taking the absolute difference between each pixel in the patches and then summing them together.

$$\sum_{[i,j] \in R} |f(i,j) - g(i,j)| \tag{C.4}$$

The point with the lowest similarity score i.e sum is the corresponding point.

C.1.1.1.5 Normalized Cross – Correlation (NCC)

Normalized cross correlation is simply cross-correlation using normalized coordinates:

$$\hat{f} = \frac{f - \bar{f}}{\sqrt{\sum (f - \bar{f})^2}} \tag{C.5}$$

$$\hat{g} = \frac{g - \bar{g}}{\sqrt{\sum (g - \bar{g})^2}} \tag{C.6}$$

$$NCC_{fg} = C_{fg}(\hat{f}, \hat{g}) = \sum_{[i,j] \in R} \hat{f}(i,j)\hat{g}(i,j) \tag{C.7}$$

NCC is the most accurate correlation measure and is invariant to the cameras intensity response characteristics or the illumination of the scene. The similarity score will be-tween -1 to 1 (perfect match). An improvement is to move away from intensity values to gradient magnitudes. The problem with NCC is it is not rotation invariant like the SSD i.e it cannot match patches that are at different orientations.

C.1.1.2 Disparity

If the left camera is located at the worlds origin and the right camera is simply a translation away then using triangle geometry the disparity of correspondences can be determined. This setup is known as simple stereo and can be seen below in figure C.2.

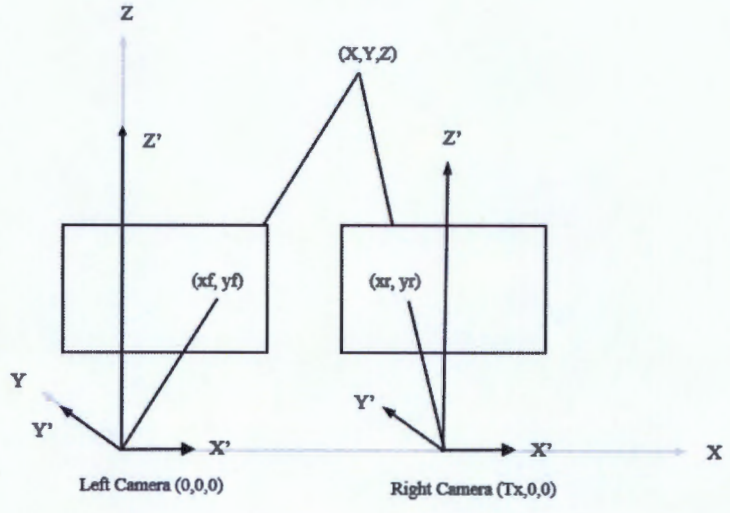


FIGURE C.2: Simple stereo: The left camera is located at the worlds origin and the right camera is simply a translation away

$$x_f = f\frac{X}{Z} \tag{C.8}$$

$$y_f = f\frac{Y}{Z} \tag{C.9}$$

$$x_r = f\frac{X - T_x}{Z} \tag{C.10}$$

$$y_r = f\frac{Y}{Z} \tag{C.11}$$

Disparity d is the image distance between corresponding image points:

$$d = x_r - x_l \tag{C.12}$$

The following equation can be obtained by subtracting the two x coordinates.

$$Z = \frac{fT_x}{d} \tag{C.13}$$

Therefore disparity is inversely proportional to depth .

A disparity map can be formed by replacing each pixel in the right image with the disparity value. In order to do this each pixels corresponding pixel must be found. As the corresponding pixels will lie on the same scan line/row correlation measures can be performed on each pixel in that row. In order to ensure a consistency amongst matches i.e to ensure that the first match is actually the correct one a disparity space image can be computed for each epipolar line.

A disparity space image has the left scan line on the x axis and the right scan line on the y axis, the points are colored by the disparity value. If there is occlusion the pixel can be given a value from the nearest pixel preceding it. The path through the disparity space image that has the highest similarity scores (not disparity scores) is chosen.

Points closer have a greater disparity than points further away. Furthermore points closer appear to move faster than points further away. This is known as motion parallax.

C.1.2 Feature-Based Matching

Feature-based i.e descriptor matching compares attributes of features. The attributes form what is known as a descriptor and then matches are found by nearest neighbor computation in descriptor space. The images are first preprocessed by a filter to extract features that are stable under the change of viewpoint i.e the features are not effected by the foreshortening issue faced by intensity-based matching

There are various types of features which can be used such as edges, corners, lines, curves, circles and ellipses, regions and a combination of features such as Scale Invariant Feature Transform (SIFT) and Speeded Up Robust Features (SURF).

Feature based methods cannot be used on images containing objects with smooth surfaces, unless patterns of light are reprojected onto the surface of objects (structured light). These patterns will create interesting points on the surfaces that would otherwise be smooth allowing the images to be matched.

C.1.2.1 Edges

Edges are robust to changes of perspective and are easy to detect. They however suffer from occlusion. The attributes of edge elements which are used for matching can be the coordinates, orientations or intensity profile (left-right of edge elements). There are many filters that can be used to detect edge elements. The most popular filters are the Canny edge detector, the LoG Filter and the Difference of two Gaussians (DoG) filter.

C.1.2.1.1 Canny Edge Detector

The Canny edge detector is widely regarded as the best performing edge detector as it has:

1. Good edge detection.

   It responds stronger at edges than to noise i.e. the max gradient magnitudes should be representative of an edge and not a normal noisy pixel who has exaggerated the derivative calculation and has as a result a high gradient magnitude.

2. Good localization.

   The maximum response should be as close as possible to the actually edge, i.e. the pixel with the highest gradient magnitude should be as close to the edge as possible.

3. Low false positives.

   There should only be one max response in the nearby vicinity around the edge i.e there should only be one max gradient magnitude in pixels surrounding the edge.

The Canny Edge detector reduces the trade-off between smoothing and good edge localization. The kernel looks similar to the derivative of Gaussian.

The Canny edge detector has three steps:

1. Find possible edge points.

   The Canny edge detector begins by smoothing the image and computing the gradient magnitude at each pixel. If the gradient magnitude is above a certain value then the pixel is marked as a possible edge point.

2. From the possible edge points find the ones that adhere to thin edges.

   Next thin edges are ensured by applying non-max suppression. Non-max suppression basically sets pixels to zero if they are not local maxima. It begins by determining the image gradients surrounding a possible edge pixel, for the pixel to remain it must have a gradient magnitude that is greater than the gradient magnitudes of its two neighbors in the gradient direction. For example if the gradient direction is 0 degrees and the edge is in a N S direction then the gradient magnitude of the pixel must be greater than the edge pixels to its left and right. This can be seen in figure C.3.

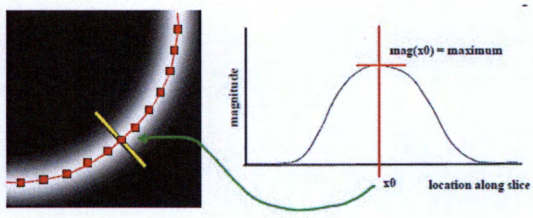


FIGURE C.3: Canny non-max suppression. Ensures the Canny detected edges are thin by preserving the points whose gradient magnitude is greater than that of their two neighbours in the gradient direction (Collins, 2007).

3. Find probably edge points.

   If threshold is too high there will be very few (if any) edges and many gaps. If the threshold is to low there will be too many edges (all pixels) and thus a large number of false positives. Hysteresis thresholding is a combination of both i.e there is a high threshold and a low threshold. Possible edge points with a gradient

magnitude above a higher threshold are kept, possible edge points with a gradient magnitude below a lower threshold are removed and possible edge points with a gradient magnitude above a lower threshold that are connected to an edge point with a gradient magnitude above a higher threshold by a path of edge points with gradient magnitudes above a lower threshold are kept.

C.1.2.1.2 LoG Filter/The Second Derivative Of Gaussian

The LoG filter involves two steps. Firstly smoothing the image with a Gaussian filter and secondly convolving the image with a Laplacian filter.

A Laplacian filter $\nabla^2 I(x, y)$ is the sum of the second derivative w.r.t to x and y.

$$\nabla^2 I(x, y) = I_{xx}(x, y) + I_{yy}(x, y) \tag{C.14}$$

It's convolution kernel is as follows:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{C.15}$$

The Laplacian is very sensitive to noise and also accentuates noise. As a result the LoG filter has been developed. The LoG filter is the combination of a Gaussian smoothing filter and the Laplacian. Hence it is the second derivative of the Gaussian.

Edges are located where the LoG filter is zero. The LoG filter detects edges which are more localized than gradient magnitudes which tend to be smeared around the edge, however with more smoothing less edges are detected. Blobs are detected at extrema of the LoG filter and are often used for gesture recognition in TV's.

The difference of two Gaussians (DoG) filter is an approximation of the Log Filter but it is much more efficient. It is the difference of two Gaussians at different scales. It is used in image coding where a laplacian pyramid stores only the difference between frames – saving on storage.

C.1.2.2 Corners

Corners are robust to changes of perspective and are easy to detect as they have large variations in the neighborhood of the point in all directions and therefore are distinctive. They however suffer from occlusion. The attributes of corners which are used

for matching can be the coordinates or type of junctions at the corners (Y-junction,L-junction,A-junction). There are many filters that can be used to detect corners. The most popular filter is the Harris corner detector . Corners can also be matched by correlation measures.

C.1.2.2.1 Harris Corner Detector

The Harris Corner Detector involves moving a window over each pixel. At each window the derivatives with regard to to x and y for each pixel are calculated using a Sobel Operator i.e I_x and I_y. These derivatives are used to populate an M matrix and are also scatter plotted.

$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{C.16}$$

An eclipse is fit to the plot, if the eigen values are both large and equal then the window is over a corner, if they are equal but small the window is over a flat region and if they are unequal the window is over an edge.

Instead of comparing two Eigen values one value can be computed called the Harris R score. The Harris R score is equal to the determinant of the M matrix minus its trace squared and scaled. A large positive score represents a corner, a large negative score represents an edge and a small score represents a flat region. The R scores of the image are then thresholded to leave behind corners /edges/flat regions (depending on threshold value). To ensure thin edges non-max suppression can be performed.

C.1.2.3 Lines

Detecting lines requires extra computation time but the benefits are that lines are more robust against occlusion as they are long and the chances of them being completely occluded are lower. The attributes of lines which are used for matching can involve the length of the line, the lines orientation, the coordinates of the midpoint, endpoints and the average intensity along the line. There are two ways to detect lines , the first involves detecting edge points and then linking and merging them to form line segments either based on similarity,collinearity or simple distance and the second way is by using the Hough Transform.

There are a two possible downfalls with matching line segments. Firstly due to image noise the end-points and mid-points of line segments are normally not reliably detected.

Every point in a line can correspond to every point on another line , the coordinates of the end points are needed to resolve this ambiguity. Secondly a line segment in one image may correspond to a broken/shorter line segment in another

C.1.2.3.1 The Hough Transform

The purpose of the Hough Transform is to detect lines in an image. The transform is robust to occlusions and isolated points. It begins by reducing the possible locations of lines in the image to the locations of edge pixels. Therefore it is assumed that edge detection has been performed and a selection of edge pixels has been found.

A line in the image (sequence of edge points) corresponds to a point in Hough Space. Lines in the image are parametrized as s'=xcos(θ) +ysin(θ) where s' is the perpendicular distance from the line to the origin and θ is the angle this perpendicular makes with the x axis. s' and θ are axis in the Hough Space. This can be seen in figure C.4.

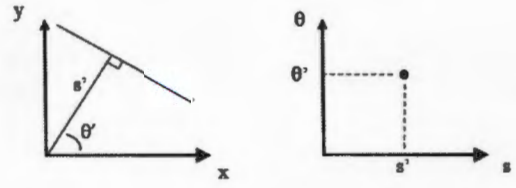


FIGURE C.4: Hough transform a) Straight line in image space. (b) Straight line in hough space.

The first step of the hough transform is to transform edge pixels (x,y) into hough space. For each edge pixel [x,y] in the image the s' value is computed for each θ between 0 and 180,one vote is then given to each of the new 180 (s', θ) positions. After transforming each edge point the points in the hough space with the highest votes correspond to the (s', θ) of lines in the image. In other words in hough space local maxima are equal to best fitting lines. The hough space has other advantages in that no clustering is needed and that the distance of points in (s', θ) is equal to the displacement of lines in (x,y).

Figure C.5 shows an image consisting of a number of lines and its equivalent representation in Hough space. In the hough space it is clear that two lines exist in the image and their corresponding (s',θ) values of these lines can be read off the axis.

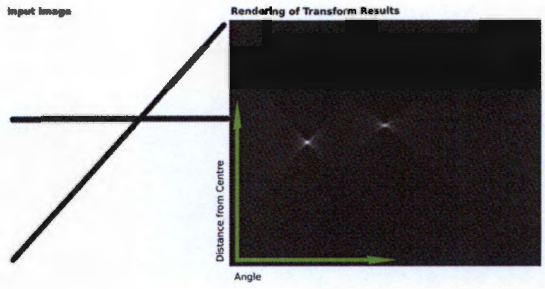


FIGURE C.5: An image consisting of two lines and there representation in Hough space (Wikipedia, 2015*b*)

Extensions to the hough transform exists such as :

1. Instead of computing s' and θ based on the image pixels they can be computed based on the image gradient
2. Giving more votes to stronger edge pixels
3. Changing the sampling of (s' , θ) to give more/less resolution
4. Applying the same procedure with circles ,ellipses ,squares or any other shape.

C.1.2.4 Regions

Regions i.e blobs can be matched. They resemble polygons and are defined by line segments. They can be detected by by intensive image segmentation algorithms such as active contour models. Their boundary does not match perfectly with other matching region boundaries in other images. Attributes which are used for matching can involve the areas of the regions, the bounding line segments of the regions and the location of the regions centroid.

C.1.2.4.1 Active Contour Models (Snakes)

Active contour models are used to perform image segmentation. Common methods such as detecting features by Hough Transform are not effective in the presence of noise or sampling artefacts (e.g. medical images) .

Active contour models look for any shape in the image that is smooth and forms a closed contour. The objects boundary is represented as a parametric curve with an energy function. The problem of finding the objects boundary is cast as an energy

minimization problem. The curve is initialized by the user close to the objects boundary, it then deforms and moves towards the desired object boundary and ultimately shrink wraps around the object, minimizing the energy. This can be seen in figure C.6.

FIGURE C.6: Active contour model - The process of shrinking a closed contour around an objects boundary (Wikipedia, 2015*a*).

The energy of the contour is comprised by internal and external components. Each of the components produces a force namely elastic and bending force for internal and external force for external. The external force acts in the direction to minimize the external energy. A newer external force is the Gradient Vector Flow (GVF), unlike the traditional external force it detects shapes with boundary concavities and has a larger capture range.

C.1.2.5 Other Features

C.1.2.5.1 Curves

Curve segments can be matched , its is an uncommon approach as every point on the curve is likely to be matched with every other point. To cater for this the attributes which are used for matching can involve the turning points.

C.1.2.5.2 Circles And Ellipses

Circles and ellipses can be matched, these are more common in indoor scenes that outdoor. They can be detected using the hough transform. Attributes which are used for matching can involve the areas in pixels or the coordinates of the center of the circle or ellipse.

C.1.2.6 Scale Invariant Feature Transform (SIFT)

In the above sections various features were explained such as edges,lines etc. How they could be detected and the attributes that are often used to describe them so that they could be matched were also discussed.

Some of these features were not invariant to scaling and rotation, meaning matching them in such cases would fail. The Scale Invariant Feature Transform (SIFT) is an algorithm to detect and describe features so that they are not affected by these complications and can be matched in all cases including occlusion (making them popular for object recognition) and noise.

The attributes used to describe the features are stored in a vector called the descriptor. In order to determine point matches these vectors are plotted in the same feature space and matches are found by nearest neighbor analysis. The dimension of the feature space is the size of the feature vector. If the ratio of the distance to the nearest neighbor to the distance of the second closest match is greater than 0.8 the match is rejected, this eliminates 80 percent of false matches and 5 percent of correct matches.

In order to detect SIFT features there are four steps:

1. Scale-space extrema detection.

   Firstly DoG's are computed at multiple scales and points that are local extrema over scale and space are found.

2. Keypoint localization.

   Secondly these potential feature points are refined by removing low contrast points and edge points (DoG has a higher response for edges). Such points are removed by thresholding the principal curvature, the principal curvature is found using the Eigen values of a Hessian matrix computed for the surrounding neighborhood. This is similar to the Harris Corner detector.

3. Orientation assignment.

   The third step is to assign an orientation to each of remaining feature points so to achieve rotation invariance. For each remaining point the gradient magnitude at its neighboring points is computed (weighted by a Gaussian kernel). Thereafter a gradient orientation histogram is plotted for this neighborhood with each pixels orientation weighted by its gradient magnitude. The highest peak in this histogram is then assigned as the orientation for the point. If there are two peaks two feature points are computed with the same location and scale but different orientations, this contributes to the stability of matching.

4. Keypoint descriptor.

   The fourth and final step is to create a feature point descriptor, this it to make the feature point invariant to remaining variations such as illumination and view point (up to 50 degrees). The feature point descriptor is formed by taking a 16x16 array of pixels around each feature point and weighting each pixels magnitude by a Gaussian, then the array is divided into 16 sub-blocks of 4x4 size. For each sub-block a 8 bin histogram is formed , the bin values are the angles between the points orientation and the orientation of the keypoint assigned in the previous step,the bin values are used to populate a 128 element feature vector, the vector is normalized to unit length to improve invariance to affine changes in illumination and the values are thresholded. The greater the size of the descriptor the better however too big increases the risk of occlusion and distortion.

The problems of SIFT features are that they are not fully affine invariant and do not work on objects that have a large illumination change (non-rigid deformations are particularly weak to large illumination change)

C.1.2.7 Speeded Up Robust Features (SURF)

SURF is an improvement on SIFT in robustness and efficiency. It is similar to SIFT in that the attributes used to describe the features are stored in a vector and that matches are found by comparing the vectors in the same manner.

It is fast as it uses integral images. An integral image is an image in which the value of a pixel at each point is the sum of the pixels above and to the left of that point.

SURF applies a Hessian matrix to each image at all points at varying scales σ. SURF feature points are blobs which are found when the determinate of the Hessian matrix is at a maximum.

The Hessian matrix at a point p is defined as :

$$H(p,\sigma) = \begin{bmatrix} l_{xx}(p,\sigma) & l_{xy}(p,\sigma) \\ l_{xy}(p,\sigma) & l_{yy}(p,\sigma) \end{bmatrix} \tag{C.17}$$

where l_{xy} is the convolution of the image with a Gaussian filter and then the second order derivative filter in the direction xy. Similarly for l_{xx} and l_{yy}. The filters are then approximated to reduce the computational cost of applying the Hessian matrix. The approximated filters are d_{xx} , d_{xy} and d_{yy}.

The Hessian matrix is further approximated by applying a relative weight w so to balance the expression of the Hessian determinant.

Therefore the matrix becomes:

$$H(p,\sigma) = \begin{bmatrix} d_{xx}(p,\sigma) & wd_{xy}(p,\sigma) \\ wd_{xy}(p,\sigma) & d_{yy}(p,\sigma) \end{bmatrix} \tag{C.18}$$

The optimal value of the relative weight w is 0.912 and therefore the determinant at a point p is defined as :

$$\det(H) = d_{xx}d_{yy} - (0.9d_{xy})^2 \tag{C.19}$$

As mentioned SURF feature points are found at maxima in the determinant values. Once SURF feature points have been found the next step is to describe them by a descriptor.

To determine the orientation of a SURF feature point the surrounding neighborhood is used. The Haar wavelet responses in the x and y directions within a circular neighborhood with radius 6s are calculated for different scales . s is the scale at which the SURF feature point was detected. The wavelet responses are weighted by a second order Gaussian with $\sigma = 2s$ and are represented in a coordinate system centered at the SURF Feature point with the horizontal and vertical directions aligned to the image coordinate system. The sum of all responses with a 60 degree sliding window are calculated and the window position with the greatest sum of responses is the dominant direction thus orientation of the SURF feature point.

The next step is to build a descriptor for each SURF feature point, a 64 dimensional descriptor is formed by placing a 16x16 quadratic grid over each SURF feature point. The grid is aligned to the orientation of the SURF feature point which was estimated above. For each of the 16 sub-regions the x,y response of the Haar wavelet filters are calculated to get a vector located at the center of each square. The vector has two components ρx and ρy.

For each sub-region the vector components are used to determine the following four values: $\sum \rho x_i, \sum \rho y_i, \sum |\rho x_i|, \sum |\rho y_i|$. These values are used as fields in the SURF feature point descriptor. Therefore the descriptor vector will have 64 values , four values for each of the 16 sub regions.

C.1.3 Position (Optical Flow) Matching

If the motion of the camera between images matched is relatively small such as between video frames then the scale and orientation of matching keypoints will not vary too much between frames and the reliance on robust descriptors for matching can be reduced and instead matches can be found as the nearest points to the predicted position of the matching point. The predicted position can be found by computing optical flow at each key point in the one image. Optical flow is explained in Appendix A. There is a clash between optical flow and baseline distance as optical flow requires small baselines and small baselines have decreased intersection angles i.e reconstruction accuracy.

C.2 Epipolar Geometry

Two overlapping images of a scene are related by epipolar geometry. There are three geometric entities involved in epipolar geometry namely the epipole , the epipolar plane and the epipolar line. This can be seen below in figure C.7

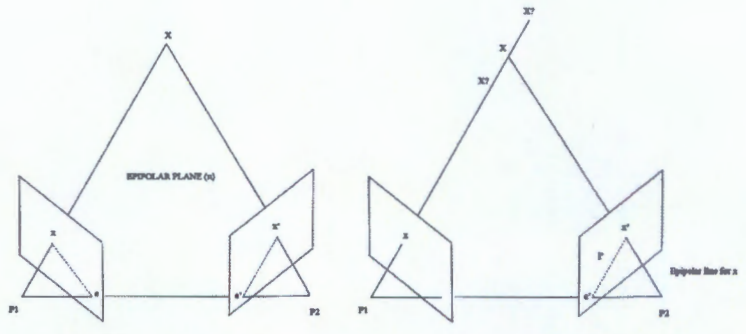


FIGURE C.7: Epipolar Geometry between two overlapping images a) The scene point X , the camera centers P1 and P2 and image points x and x' lie in a common plane π called the epipolar plane b) The ray from the first camera P1 is imaged as a line on the second image called the epipolar line. The scene point X which projects onto the second image lies on this line. The same exists for the ray from the second camera

The epipole is the point of intersection between the baseline (the line joining the cameras) and the image plane. There is an epipole in each image . The epipole in each image is the location of the camera of the other view.

The epipolar plane is the plane containing the baseline and the scene point. There is an epipolar plane for each scene point and so the epipolar planes "rotate" about the baseline. This family of planes is known as an epipolar pencil as seen below in figure C.8.

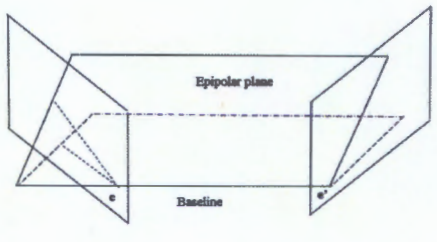


FIGURE C.8: Epipolar pencil. The collection of epipolar planes which share the same baseline

The epipolar line is the intersection of an epipolar plane with the image plane. There is en epipolar line for each epipolar plane and all epipolar lines intersect at the epipole.

The fundamental matrix represents the epipolar geometry algebraically. It is a 3x3 matrix and therefore has 9 elements. 8 of these elements are independent as the common scaling is not significant . Furthermore because F satisfies the constraint detF = 0 it has 7 degrees of freedom. As it represents the geometry it can be used to determine the rotation and translation of the camera between images. This is the main use of the fundamental matrix and will be discussed in the next section.

C.2.1 The Epipolar Constraint

As can be seen in figure C.7 the reprojection of a ray from the camera center to an image point in one image to the other image is the epipolar line ,therefore there is a mapping between image point and epipolar line .

This mapping can be decomposed into two steps:

1. Point transfer via a plane.

   The ray through point x in the first image meets at a plane π at X. This point is then projected to the second image at point x'. The points x and x' are both images of the 3D point X lying on the plane. The set of each image points are projectively equivalent as they are projectively equivalent to the planar point set X_i. Therefore there is a homography H_π mapping each x_i to x'_i.

   $$x' = H_\pi x \tag{C.20}$$

2. Constructing the epipolar line.

   The epipolar line l' passing through point x' can be written as:

   $$l' = [e']_x x' \tag{C.21}$$

   Where :

   $$e' = (\ a_1 \quad a_2 \quad a_3)^T \tag{C.22}$$

   $$[e']_x = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{C.23}$$

Owing to the homography H_π this can be rewritten as:

$$l' = [e']_x H_\pi x \tag{C.24}$$

Making :

$$F = [e']_x H_\pi \tag{C.25}$$

These equations can be rewritten as:

$$l' = Fx \tag{C.26}$$

$$l = F^T x' \tag{C.27}$$

Therefore this mapping is represented by the fundamental matrix

This means that image points must lie on corresponding epipolar lines. This is known as the epipolar constraint and can be be used to find more matches by defining the search space (epipolar line) in the second image corresponding to each point in the first image. For each point x in one image the epipolar line is Fx in the second image. The corresponding point is not guaranteed to lie on the line Fx as the fundamental matrix is only known within certain bounds which are illustrated by its covariance matrix.

Therefore as the corresponding point is not guaranteed to lie on the line Fx it is probably going to lie in the space on each side of the line and therefore this space is what needs to be searched. To determine this space the covariance matrix of the fundamental matrix $\sum_F$ is determined and then transfered to a covariance matrix of the epipolar line $\sum_l$. If epipolar lines are normally distributed which is not always the case, then the covariance matrix $\sum_l$ and the mean $\overline{l}^T$ of the epipolar line can be used to determine the plane conic with a probability α that the lines will lie in this region.

$$C = \overline{l}\overline{l}^T - k^2 \sum_l \tag{C.28}$$

Where

$$F_n^{-1}(k^2) = \alpha \tag{C.29}$$

(n=2 since the covariance matrix $\sum_l$ has rank 2)

C.2.2 Determining The Fundamental Matrix From Point Correspondences

The fundamental matrix can be determined from point correspondences x and x'.

$$x = (x, y, 1)^T \tag{C.30}$$

$$x' = (x', y', 1)^T \tag{C.31}$$

The most basic property of the fundamental matrix is that it satisfies the correspondence condition (epipolar constraint):

$$x'^T F x = 0 \tag{C.32}$$

Therefore :

$$\mathrm{x'xf_{11}}+\mathrm{x'yf_{12}} + \mathrm{x'f_{13}} + \mathrm{y'xf_{21}} + \mathrm{y'yf_{22}} + \mathrm{y'f_{23}} + \mathrm{xf_{31}} + \mathrm{yf_{32}} + \mathrm{f_{33}} = 0 \tag{C.33}$$

$$(x'x, x'y, x', y'x, y'y, y', x, y, 1)f = 0 \tag{C.34}$$

$$Af = 0 \tag{C.35}$$

This is a homogeneous set of equations and f can only be determined up to scale. As f has 8 degrees of freedom a minimum of 8 constraints are needed to solve for f. There are various methods for solving F depending on the number of point correspondences such as the normalized 8-point algorithm.

An important property of the fundamental matrix is that it is singular, in fact it is rank 2. If the fundamental matrix is not singular then the computed epipolar lines will not be coincident. This can be seen in figure C.9. A singular matrix has a determinant of zero and is not invertible. The matrix F found by solving linear equations is often not rank 2 and steps often have to be enforced to enforce this constraint.

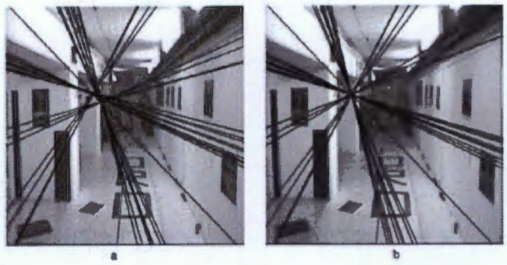


FIGURE C.9: Epipolar lines a) The effect of a non-singular fundamental matrix - the epipolar lines do not meet in a common epipole b) The effect of enforcing singularity using SVD (Collins, 2007).

C.2.2.1 The Normalized 8-Point Algorithm (8 Or More Points)

As there are 8 degrees of freedom if there are 8 point correspondences the solution will be unique and equal to the generator of the right null space of A.

If there are more than 8 point correspondences the system will be over constrained and the least squares solution will be required.

There are three steps in determining the least squares solution:

1. Normalization

   It is very important that the points are normalized before the equations to solve are formulated. To normalize the points each image is translated and scaled so that the centroid of reference points is at the origin of the coordinates and the RMS distance of the points from the origin is equal to $\sqrt{2}$. T and T' are normalizing transformations consisting of a translation and scaling

$$'\hat{x}_i = T\hat{x}_i \tag{C.36}$$

$$\hat{x}_i = T'\hat{x}'_i \tag{C.37}$$

2. Find linear solution for F

   The least squares solution for f is the singular vector which corresponds to the smallest singular value of A. The smallest singular value of A is the last column of V in the SVD $A = UDV^T$. The vector f found minimizes $||Af||$ subject to the condition $||f||$=1. The vector f makes up F.

3. Constraint enforcement

   As mentioned in most cases the fundamental matrix will not have rank 2 and extra steps need to be followed to enforce the singularity constraint and make the matrix rank 2 , such as correcting the matrix F using the SVD solution of A. F is replaced by the matrix F' that minimizes the Frobenus norm $||F - F'||$ subject to the condition detF'=0. The Frobenus norm is the square root of the sum of the absolute squares of its elements.

   To calculate matrix F' the following steps are followed:

   (a) SVD $F = UDV^T$ where D is a diagonal matrix D = diag(r,s,t) where $r > s > t$.

   (b) F' = Udiag(r,s,0)V^T

4. Denormalization

   After matrix F' has been calculated the matrix is denormalized so that it corresponds to the original data.

$$F = T'^T \widehat{F}' T \tag{C.38}$$

C.2.2.2 The 7-Point Case (7 points)

If there are 7 point correspondences the A matrix will generally be of rank 7 which means there will be 7 independent equations/constraints. The solution can be determined however as there are 8 degrees of freedom and only 7 equations/constraints the singularity constraint detF=0 is used.

The solution to the equation Af=0 is a 2-dimensional space of the form F= αF1+(1-α)F2, where α is a scalar. F1 and F2 are the matrices corresponding to f1 and f2 which are generators of the right null-space of A.

The singularity constraint is written as det(αF1+(1-α)F2)=0. As F1 and F2 are known a cubic polynomial equation in α is formed. There are either one or three solutions for α and therefore as F = αF1+(1-α)F2 there are one or three possible solutions for F.

If the 7 point correspondences and the two camera centers lie on a ruled quadric this is a degenerate configuration and there will be three possible solutions for F. There will be one solution for F if they lie on a non-ruled quadric.

C.2.2.3 Automatically Using The 7-Point Case (7 points)

The following steps are followed to automatically compute the fundamental matrix using the 7 point case:

1. Find keypoints. Find keypoints in each image
2. Match keypoints. Determine matches between keypoints
3. Refine matches using RANSAC.
   (a) Select a random sample of 7 correspondences and compute the fundamental matrix using the 7 point case. There will be one or 3 solutions.
   (b) Calculate the distance d for each potential correspondence to the corresponding epipolar line.
   (c) Compute the number of inlier's consistent with the fundamental matrix. This is equal to the number of matches for which d<5.99σ pixels
   (d) If there are 3 solutions for F the number of inlier's must be determined for each solution.
   (e) Repeat for N samples and choose the F with the largest number of inlier's. The choice of N depends on the probability of no outliers desired.
4. Optimal estimation.

   Re-estimate F from all correspondences classified as inlier's.
5. Assisted matching.

   More matches can be found now that F can be used to define a search region in the alternate image i.e the epipolar line can be searched in the alternate image for a match for each image point x. Therefore weaker similarity thresholds can be used.

The advantages of using 7 point correspondences are firstly the fundamental matrix is inherently of rank 2 and no constraints have to be used to enforce this rank and secondly the N number of samples selected for a 99 percent confidence of no outliers is half the size in comparison to when 8 correspondences are used. The only disadvantaged of using 7 correspondences is that three solutions for F may be present.

C.2.2.4 Special Cases Of Determining The Fundamental Matrix

C.2.2.4.1 Noisy Point Coordinates

If the rank of the matrix is 9 this means that the data is not exact i.e that there is noise in the point coordinates . A rank of 9 also results in 9 equations/constraints. As there are only 8 degrees of freedom (dof) the system is therefore over constrained and a least squares solution is found.

The precision of the epipolar lines corresponding to matched points can also be weak and what is known as an envelope of possible epipolar lines arises. The more matched points used in determining the fundamental matrix the more precise the epipolar line is and the narrower the envelope is. The max precision achievable depends ultimately on the accuracy of the matching.

C.2.2.4.2 Pure Translation

This is the simplest special case and involves only translation and no rotation.

The scene points move parallel to the camera, their image correspondences diverge at a vanishing point in the direction of motion. This vanishing point is the epipole e. The epipole is a fixed point, also known as the Focus of Expansion (FOE) and has the same coordinates in both images. This can be seen in figure C.10.

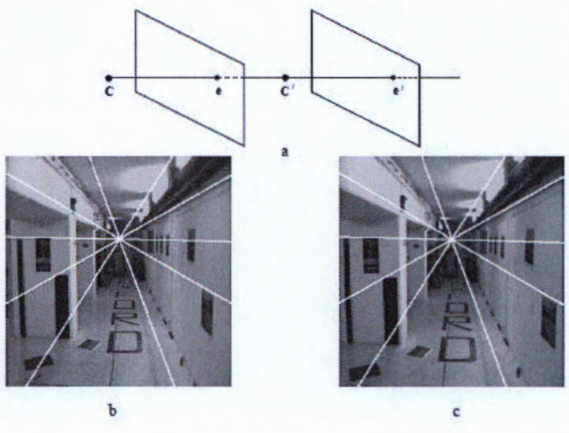


FIGURE C.10: Special case: Pure translation of the camera (a) Under pure translation the epipole is a fixed point i.e has the same coordinates in both images and points appear to radiate along outward lines. b) and c) Each image has the same epipolar lines yet in the second image the image points have slid further along the line (Collins, 2007)

Points closer to the camera appear to move faster than those further away. In pure translation both corresponding image points and the epipole are collinear . This is known as auto-epipolar.

The equation reduces to:

$$F = [e']_x \tag{C.39}$$

This matrix F satisfies the required condition that it is rank 2. For pure translation the coordinate of the epipole can be computed uniquely from two point correspondences.

It is possible to achieve pure translational motion by rotating the first camera and applying corrections for the differences in calibration matrices between the two images. The result is a projective transformation H of the two images.

$$F = [e']_x H \tag{C.40}$$

C.2.2.4.3 Pure Planar Motion

If the rotation axis is orthogonal to the direction of translation the condition det(Fs) = 0 is an additional constraint on the fundamental matrix. Fs is the symmetric part of the fundamental matrix. Therefore the degrees of freedom of the fundamental matrix are reduced from 7 to 6.

C.2.2.5 Degeneracies

Point correspondences are degenerate if they fail to uniquely define the epipolar geometry or if there exists a number of alternative fundamental matrices that satisfy the same fundamental matrix equation such as F1 and F2.

$$x'^T F1x = 0 \tag{C.41}$$

$$x'^T F2x = 0 \tag{C.42}$$

If the point correspondences are degenerate and they are used to define the fundamental matrix then the matrix is likely to be numerically ill conditioned. To avoid this the fundamental matrix can be computed accurately from the camera matrices.

There are three configurations of points which may be degenerate and thus lead to an numerically ill-conditioned fundamental matrix:

1. Points lie on a ruled quadric (Structural Degeneracy).

   A quadric is a surface in R^3 defined by the equation:

   $$X^T Q X = 0 \tag{C.43}$$

   A ruled quadric is a quadric surface that contains a straight line. If the camera centers and all 3D points lie on a ruled quadric (i.e critical surface) and Q is singular the configuration of points will be degenerate. For a critical surface configuration there are three possible fundamental matrices.

2. Points on a plane (Structural Degeneracy).

   If all the points lie in a plane they will be related by a homography i.e 2D projective transformation H , as a result there there will be a number of possible fundamental matrices for the same H, therefore the correspondences are degenerate.

3. No Translation (Motion Degeneracy).

   If the two camera centers are coincident then the epipolar geometry is not defined and the point correspondences used to determine the fundamental matrices will be degenerate. The Fundamental Matrix should be zero and the only way to get this value is if its computed from the camera matrices.

Degeneracies can be detected by looking at the dimension of the null-space f of A. Remember Af=0. The dimension of a matrix is the number of rows times columns. If

the dimension is 1 there is likely to be no degeneracy and a unique fundamental matrix. If the dimension is equal to 2 then there are likely to be 1 or 3 solutions for F which may come from degenerate point correspondences. If the dimension is equal to 3 then the point correspondences are related by homography and are definitely degenerate.

C.2.3 Random Sample Consensus (RANSAC)

Random Sample Consensus (RANSAC) is a robust estimation technique that negates the effect of outliers. It is a two stage process , firstly data points are classified as inliers and outliers and then a model is fit to the inliers (by least squares) whilst ignoring the outliers.

1. Draw a sample of S points taken from the data uniformly and randomly
2. Fit a model to the sample by least squares
3. For each data point outside the sample measure the distance to the line. Count the amount of points that are under the threshold distance d.
4. If the amount of points is more than a certain value T then there is a good fit
5. Refit the line using all these points by least squares.
6. Repeat with a different sample N times

The number of iterations N depends on the confidence of getting a pure-inlier sample eg N=5 may result in a 99 percent chance of getting a pure inlier sample. The distance threshold d is $3.84s^2$ when a line/fundamental matrix is fit to the points and $5.99s^2$ when a homography is fit to the points. s is the measurement error.

C.2.4 Determining The Fundamental Matrix From The Normalized Camera Matrices

The fundamental matrix is purely dependent on the camera matrices. If the cameras are calibrated and the first camera is positioned at the world origin the fundamental matrix can be determined as follows:

$$H_\pi = P'P^{-1} \tag{C.44}$$

$$F = [e']_x(P'P^{-1}) \tag{C.45}$$

Where :

$$e = KR^Tt \tag{C.46}$$

$$e' = K't \tag{C.47}$$

This is the most accurate way of determining the fundamental matrix as it is unaffected by noise in point correspondences.

C.2.5 Determining The Accuracy Of The Fundamental Matrix

The residual error is :

$$\frac{1}{N}\sum_i^N d(x'_i, Fx_i)^2 + d(x_i, F^Tx'_i)^2 \tag{C.48}$$

d(x,l) is the distance in pixels between a point in one image and the epipolar line in that image, on which it is supposed to lie.

The error is averaged over all N correspondences and not just the n correspondences which are used to calculate the fundamental matrix.

C.2.6 The Essential Matrix

The Essential matrix is simply the fundamental matrix computed using normalized points.

$$\hat{x}'^T E \hat{x} \tag{C.49}$$

To get normalized coordinates the coordinates can be multiplied by the inverse of the calibration matrix or a normalized camera matrix can be used.

$$x' = K^{-1}x \tag{C.50}$$

$$x' = [R|t]X \tag{C.51}$$

The essential matrix has 5 degrees of freedom. The reduced number of degrees of freedom in comparison to the fundamental matrix results in more constraints.

C.2.6.1 Determining The Essential Matrix From the Camera Matrices

For a pair of normalized camera matrices P = $[I|0]$ and $P' = [R|t]$ the essential matrix is equal to the cross product of the translation t and rotation matrix R.

$$E = [t]_x R \tag{C.52}$$

C.2.6.2 Determining The Essential Matrix From the Fundamental Matrix

The essential matrix may also be computed from the fundamental matrix F if the calibration matrix K is known.

$$E = K'^T F K \tag{C.53}$$

C.2.6.3 Determining The Essential Matrix From Point Correspondences

C.2.6.3.1 The 5-point case (5 points)

The essential matrix can be computed from 5 points if the calibration matrix of the camera is known.

There are three key steps:

1. Determine the nullspace of a 5x9 matrix.

   Each of the five point correspondences give rise to the below constraint

$$q'^T E q = 0 \tag{C.54}$$

   q' and q are the normalized points.

   The above constraint is rewritten as

$$\tilde{q}^T \tilde{E} = 0 \tag{C.55}$$

   where

$$\tilde{q} = \begin{bmatrix} q_1 q'_1 & q_2 q'_1 & q_3 q'_1 & q_1 q'_2 & q_2 q'_2 & q_3 q'_2 & q_1 q'_3 & q_2 q'_3 & q_3 q'_3 \end{bmatrix} \tag{C.56}$$

$$\tilde{E} = \begin{bmatrix} \tilde{E}_{11} & \tilde{E}_{12} & \tilde{E}_{13} & \tilde{E}_{21} & \tilde{E}_{22} & \tilde{E}_{23} & \tilde{E}_{31} & \tilde{E}_{32} & \tilde{E}_{33} \end{bmatrix}^T \tag{C.57}$$

   The above vectors $\tilde{q}$ for each correspondence are stacked forming a 5x9 matrix. The four vectors X' Y' Z' and W' that span the right null space are then computed by QR-factorization. The vectors correspond directly to four 3x3 matrices and the essential matrix must be of the form:

$$E = xX + yY + zZ + wW \tag{C.58}$$

   x,y,z and w are scalars.

2. Expansion of the cubic constraints.

   The cubic constraints are:

$$\det(F) = 0 \tag{C.59}$$

$$EE^T E - \frac{1}{2} trace(EE^T)E = 0 \tag{C.60}$$

   The essential matrix must satisfy the above equations/constraints. The essential matrix equation E is inserted into the 10 cubic constraints.

3. Gauss-Jordan elimination.

   The variables x ,y and z are found from Gauss Jordan elimination and used to obtain the essential matrix.

The advantages of determining the essential matrix using the 5 point algorithm is that it is more robust to noise than the 7 and 8 point algorithms and is more accurate when the camera transverses in sideways motion. Furthermore it is robust to planar degeneracies. It however leads to weaker accuracies than the 8 point matrix when the camera transverses in forward motion .

C.2.7 Image Rectification

Image rectification is the process of transforming two overlapping images so that only pure translational motion exists between them i.e the epipolar lines are parallel with the x-axis and thus disparities between images are in the x direction only . This can be seen in figures C.11 and C.12. Image rectification also removes perspective distortion.

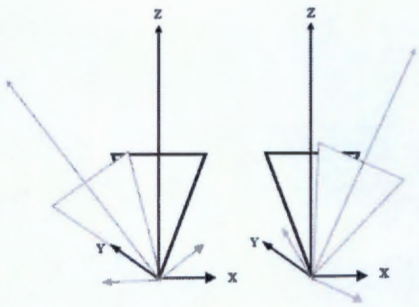


FIGURE C.11: Image rectification. The images have been rotated so that their planes are parallel with the baselines

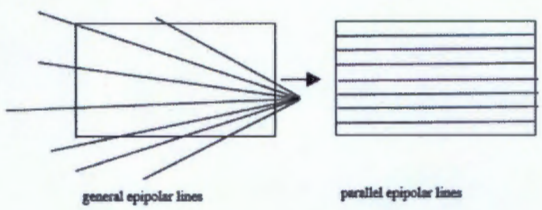


FIGURE C.12: Image rectification. The epipolar lines before and after rectification. After rectification they are parallel.

There are two approaches to image rectification. The first approach (Approach A) requires the rotation and translation of the camera to be known , whereas the second approach (Approach B) does not.

C.2.7.1 Approach A

The first approach to image rectification assumes the first normalized camera matrix is of canonical form $[I|0]$ and determines the rotation and translation between the next camera from the camera matrix of the second camera, which can be determined from the essential matrix as shown in the above section.

The rectification involves four steps:

1. Build the rectification matrix. This matrix maps the epipoles in the first image to infinity.

$$Rrect = \begin{bmatrix} {e_1}^T \\ {e_2}^T \\ {e_3}^T \end{bmatrix} \tag{C.61}$$

where:

$$e_1 = \frac{T}{\|T\|} \tag{C.62}$$

$$e_2 = \frac{1}{\sqrt{T_x^2 + T_y^2}} \begin{bmatrix} -T_y \\ T_x \\ 0 \end{bmatrix} \tag{C.63}$$

$$e_3 = e_1 \times e_2 \tag{C.64}$$

2. Set R_l = Rrect and R_r = R*Rrect.

3. For each point in the left image p_l do the following:

$$p_l = (x, y, f)^T \tag{C.65}$$

$$R_l p_l = (x', y', z')^T \tag{C.66}$$

$$p_l{}' = \frac{f}{z'}(x', y', z')^T \tag{C.67}$$

4. Repeat the above for each point in the right image p_r using R_r.

C.2.7.2 Approach B

In the second approach the rotation and translation of the camera is not known. So the 2D projective transformation H' which rectifies an image i.e maps the epipole to infinity is sought. If an inappropriate H 'is chosen severe projective distortions of the image can take place. A value of H' which is often leads to good results is one in which acts as a rigid transformation (just rotation and translation) in the neighborhood of a given selected point in the image.The given point is often the center of the image x_0.

$$H' = GRT \tag{C.68}$$

Where T is the translation taking the point x_0 to the origin and R is the rotation about the origin taking the epipole e to a point on the x axis (f,0,1) and where G is the transformation which maps the epipole e to the point of infinity.

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix} \tag{C.69}$$

To match up epipolar lines in two images a transformation needs to be performed on the other image (i.e matching transformations). The first image has been rectified i.e H' has been applied to the image mapping the epipole's to infinity and bringing the epipolar lines parallel to the x axis . The matching transformation H to be applied to the other image will be chosen as to minimize the square distances between matched points x_i and x_i'.

$$\sum_i d(Hx_i, H'x_i')^2 \tag{C.70}$$

The output is a pair of images re-sampled so that there epipolar lines are horizontal (parallel with the x axis). This means any remaining disparity between matching points will be along the horizontal epipolar lines i.e any point in the first image will now match a point in the second image with the same y coordinate. The benefit of this is that if any additional matching points are sought the area to be searched is confined to 1 dimension.

The transformation H of the second image matches the transformation H' of the first image (which sends the epipole to infinity) if and only if H = $H_A H_0$ where $H_0 = H'M$ and H_A is an affine transformation. M is the transformation by the plane π

$$H_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C.71}$$

H_A is determined such that the following equation is minimized:

$$\sum_i d(H_A H_0 x_i, H' x_i')^2 \tag{C.72}$$

As H_A is affine simple linear-least squares can be used.

Letting $\hat{x}_i = (\hat{x}_i, \hat{y}_i, 1)^T$ and $\hat{x}_i' = (\hat{x}_i', \hat{y}_i', 1)^T$ the above equation can be written as:

$$\sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}_i')^2 + (\hat{y}_i - \hat{y}_i')^2 \tag{C.73}$$

C.3 The Camera Matrices

The camera matrix is made up of the calibration matrix and the extrinsic parameters rotation and translation . The rotation is the rotation of the worlds coordinate system in the camera coordinate system and the translation is the location of the worlds origin in the cameras coordinate system. This is the opposite of the camera pose.

The standard camera matrix

$$P = K[R|t]x \tag{C.74}$$

The normalized camera matrix (the extrinsic parameter matrix):

$$K^{-1}P = [R|t] \tag{C.75}$$

The standard coordinate:

$$x = PX \tag{C.76}$$

A pair of camera matrices determines a unique fundamental matrix, however pairs of camera matrices that differ by a projective transformation give rise to the same fundamental matrix. Therefore the fundamental matrix captures the projective relationship of the two cameras.

C.3.1 Determining The Normalized Camera Matrix From The Fundamental Matrix

As seen in the section before the fundamental matrix can be determined from the camera matrices. The opposite is also true and the camera matrices can be determined from the fundamental matrix , up to a projective ambiguity.

If the first normalized cameras matrix is $P = [I|0]$ the second camera matrix is determined as:

$$P' = \left[[e']_x F + e' v^T | \lambda e'\right] \tag{C.77}$$

where v is the position of the plane at infinity $\mathrm{v} = [v_x, v_x, v_x]^T$ and λ is a non-zero scalar which determines the global scale of the reconstruction.

Now that both normalized camera matrices are known a stratified reconstruction approach can be followed to upgrade from a projective reconstruction to a metric/similarity reconstruction.

C.3.2 Determining The Normalized Camera Matrix From The Essential Matrix

If the first normalized cameras matrix is $P = [I|0]$ and the cameras are calibrated then camera matrix of the second camera can be obtained from the essential matrix and a similarity/metric reconstruction can be achieved directly.

In contrast with the fundamental matrix where there is a projective ambiguity when the essential matrix is used the cameras matrices may be determined up to a scale and fourfold ambiguity i.e when the essential matrix is singular value decomposed there are four possible solutions for the second camera P' as seen below:

$$E = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{C.78}$$

$$P' = [R|T] \tag{C.79}$$

$$P' = [UWV^T|u_3]/[UWV^T|-u_3]/[UW^TV^T|u_3]/[UW^TV^T|-u_3] \tag{C.80}$$

A reconstructed point X will be in front of both cameras in one of these four solutions only. Therefore a single point is necessary to test and determine the correct solution. This can been seen in figure C.13.

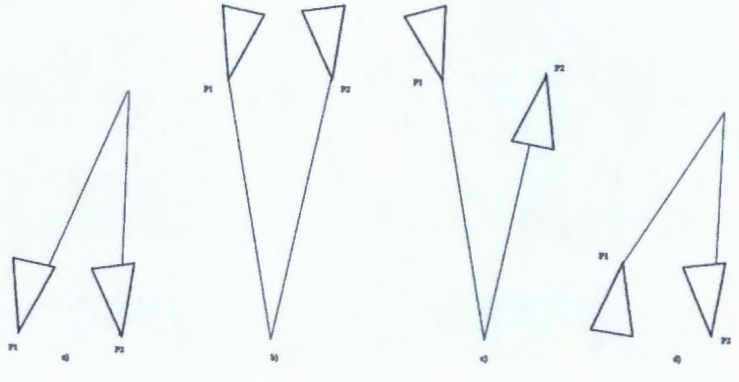


FIGURE C.13: Essential matrix decomposition ambiguity. The four possible solutions for the second camera matrix . (a) Is the correct solution as the reconstructed point is in front of both the cameras

The camera matrix determined is a normalized camera matrix. Therefore the rotation and translation of the camera can be read from the matrix. This matrix only works with normalized coordinates and must be multiplied by the calibration matrix to determine the normal camera matrix.

C.3.3 Determining The Normalized Camera Matrix From 3D-2D Correspondences (PNP)

The normalized camera matrix can be determined from 3D-2D correspondences in similar fashion to camera calibration using a 3D object which is described in in appendix A.

C.4 Triangulation

Triangulation is the process of determining the scene coordinate X from two corresponding image coordinates x and x'. It is assumed the camera matrices P and P' and hence the fundamental matrix is known or that the fundamental matrix is known and thus the camera matrices can be determined from it.

The two camera rays which go through the image coordinates x and x' should lie in a common epipolar plane i.e a plane passing through the two camera centers and therefore should intersect. However in general the rays will not intersect due to errors in the measured image coordinates. Therefore the rays will be skew as can be seen in figure C.14.

This means there will not be a point X with satisfies the below equations exactly:

$$x = PX \tag{C.81}$$

$$x' = P'X \tag{C.82}$$

This is because the image points do not exactly satisfy the epipolar constraint $x'^T Fx = 0$ exactly.

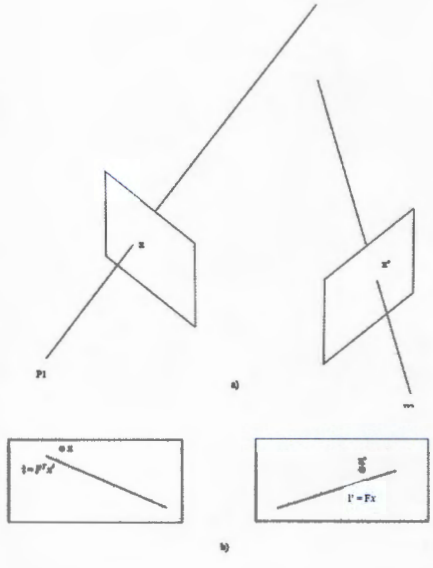


FIGURE C.14: The epipolar constraint a) Two skew rays which will not intersect as their image points do not satisfy the epipolar constraint b) The image of the ray through x is the line l'=Fx , since the rays do not intersect x' does not lie on l', and vice versa.

Points on the baseline can not be determined as there camera rays are collinear and intersect along their whole length.

There are various triangulation methods which attempt to give the best estimate of the location of the scene point X. If the camera matrices are only known up to say a projective transformation H then a triangulation method that finds the scene point X at the midpoint of the common perpendicular to the two rays in space is not suitable as distance and perpendicularity are not valid in the context of projective geometry. This method is therefore not projective invariant.

So an ideal triangulation method τ used must be invariant to a transformation H of the space such as:

$$X = \tau(x, x', P, P') = H^{-1}\tau(x, x', PH^{-1}, P'H^{-1}) \tag{C.83}$$

Furthermore an ideal triangulation method determines the scene coordinate X which exactly satisfies the supplied camera geometry.

$$x = PX \tag{C.84}$$

$$x' = P'X \tag{C.85}$$

C.4.1 Simple Triangulation

The first triangulation method is a simple linear triangulation method. It is not projective invariant and the estimated scene point X does not exactly satisfy the existing supplied camera geometry.

The two geometry equations x = PX and x' = P'X are combined into the form

$$AX = 0 \tag{C.86}$$

To get the equations into this form the following steps are followed :

1. For each image the scale factor is eliminated by computing the cross product of itself, x x x = 0 i.e x x (PX) = 0. This results in three equations for each image.

$$x(p^{3T}X) - (p^{1T}X) = 0 \tag{C.87}$$

$$y(p^{3T}X) - (p^{2T}X) = 0 \tag{C.88}$$

$$x(p^{2T}X) - y(p^{1T}X) = 0 \tag{C.89}$$

   As can be seen the last equation is linearly dependent on the first two.

2. The A matrix is then populated with the four linearly independent rows (two from each image)

$$A = \begin{bmatrix} xp^{3T} - p^{1T} \\ yp^{3T} - p^{2T} \\ x'p'^{3T} - p'^{1T} \\ y'p'^{3T} - p'^{2T} \end{bmatrix} \tag{C.90}$$

There are two ways of solving this equation for X: The Homogeneous method and the Inhomogeneous method.

C.4.1.1 Homogeneous Method (Direct Linear Transform)

The Homogeneous method solves for X based on the homogeneous equation

$$AX = 0 \tag{C.91}$$

There are four equations and three unknowns (XYZ). Therefore there are more equations than unknowns and this is an overdetermined system . In general it has no non-zero solution. A solution which is as close to X is found i.e a solution that minimizes $||AX||$, this is the least squares solution.

If X is a solution to this set of equations, then so is KX for any scalar k. To find one solution X the constraint $||X||$=1 is enforced. To find the X that minimizes $||AX||$ subject to this constraint singular value decomposition is performed:

$$A = UDV^T \tag{C.92}$$

X is the unit singular vector corresponding to the smallest singular value of A. i.e X is the last column of V.

C.4.1.2 Inhomogeneous Method

The inhomogeneous method solves X based on the equation:

$$AX = b \tag{C.93}$$

To get the equations into this non-homogeneous form X is set to:

$$X = (X, Y, Z, 1)^T \tag{C.94}$$

There are four equations and three unknowns (XYZ). Therefore there are more equations than unknowns and this is an overdetermined system . The system may not have a solution and thus a solution which is as close to X is found i.e a solution that minimizes $||AX - b||$, this is the least squares solution.

$$A = UDV^T \tag{C.95}$$

$$b' = U^T \tag{C.96}$$

$$y_i = b'_i / d_i \tag{C.97}$$

Where d_i is the i'th diagonal of entry of D.

The solution is then X = Vy.

As the last homogeneous coordinate of X is 1 it is assumed the true solution for X does not have a last coordinate close to 0 (the point is at infinity), if so instabilities will arise. The disadvantage therefore of this method is that it is unsuitable for points which lie on the plane at infinity.

C.4.1.3 Discussion

Both of methods of solving for X are not projectively invariant , in other words if the cameras go through a projective transformation H the X's will not correspond. This can be seen in the example below:

Suppose camera matrices P and P' are transformed by H.

- The P and P' matrices will be replaced by PH^{-1} and $P'H^{-1}$
- The A matrix will become AH^{-1}.
- The point X will become HX.
- Before transformation AX = error.
- After transformation $AH^{-1}HX$ = error

Therefore there is a one to one correspondence between points X and HX which give the same error. However neither the constraint $||X|| = 1$ (for the homogeneous method) and the constraint X= $(X, Y, Z, 1)^T$ (for the inhomogeneous method) are invariant under the application of a projective transformation. Therefore in general the reconstructed point X will not correspond to the point HX which solves the transformed problem.

If the reconstruction is affine the situation is different. The homogeneous constraint $||X|| = 1$ is not preserved, however the inhomogeneous constraint X= $(X, Y, Z, 1)^T$ is preserved. Therefore the inhomogeneous method is affine invariant.

The next triangulation method discussed will be invariant to the projective frame of the cameras and minimize a geometric image error.

C.4.2 Optimal Triangulation

The method involves finding the point X which exactly satisfies the supplied camera geometry. This point is the point which minimizes the reprojection error .

The reprojection error is the (summed square) distance between the reprojections of X onto the images $\hat{x}$ and $\hat{x}'$ and the measured image points x and x'. The noise in the image measurements is assumed to be Gaussian.

We therefore look for the points $\hat{x}$ and $\hat{x}'$ that minimize the function:

$$C(x, x') = d(x, \hat{x})^2 + d(x', \hat{x}')^2 \tag{C.98}$$

subject to the constraint:

$$\hat{x}'^T F \hat{x} = 0 \tag{C.99}$$

This is known as the minimization problem.This method results in what is known at the maximum likelihood estimate. This method is projectively invariant as only the image distances are minimized and the image distances do not depend on whether the reconstruction was projective , affine or metric/similarity.

Once the maximum likelihood points i.e $\hat{x}$ and $\hat{x}'$ are found the point X may be found using any triangulation method as the points are guaranteed to meet precisely in space.

To find the maximum likelihood points that minimize the function the following steps are followed:

1. Reformulate the minimization problem.
   (a) Point correspondences x and x' have been measured and the Fundamental Matrix F is known(which hopefully has been computed from the camera matrices to avoid noise from point correspondences effecting its accuracy)
   (b) The optimum point $\hat{x}$ will lie on the epipolar line l and the corresponding optimum point $\hat{x}'$ will lie on the epipolar line l' .
   (c) Any other pair of points lying on these lines will also satisfy the epipolar constraint. The point on the lines closest to the measured points is the points $x_\perp$ which are perpendicular to the measured points. Therefore $\hat{x} = x_\perp$ and $\hat{x}' = x'_\perp$.
   (d) The minimization problem can then be reformulated as:

$$d(x,l)^2 + d(x',l')^2 \tag{C.100}$$

Where l and l' range over all the choices of corresponding epipolar lines. This is known as the distance function

2. Parameterize the minimization problem.

   (a) First the pencil of epipolar lines in the first image is parameterized by a parameter t i.e l(t) and l'(t)

   (b) Next the fundamental matrix F is used to compute the corresponding epipolar line l'(t) in the second image

   (c) The distance function is then expressed as a function of t

$$d(x,l(t))^2 + d(x',l'(t))^2 \tag{C.101}$$

3. Find the value of t which minimizes the function.

   (a) The first step is to apply a rigid transformation to each image so that the points x and x' move to the origin $(0,0,1)^T$. This rigid transformation has no effect on the distance function

$$T = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \tag{C.102}$$

$$T' = \begin{bmatrix} 1 & 0 & -x' \\ 0 & 1 & -y' \\ 0 & 0 & 1 \end{bmatrix} \tag{C.103}$$

   (b) Replace F by $T'^{-T}FT^{-1}$

   (c) Determine the coordinates of the right and left epipoles $e = (e_1, e_2, e_3)^T$ and $e' = (e_1', e_2', e_3')^T$ such that $e'^T F = 0$ and $Fe = 0$.

   (d) Normalize the coordinates of the epipoles such that $e_1^2 + e_2^2 = 1$. Do the same for e'

   (e) Form the following matrices which will move the epipoles to the x axis at points $(1,0,f)^T$ and $(1,0,f')^T$ respectively.

$$R = \begin{bmatrix} e_1 & e_2 & 0 \\ -e_2 & e_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C.104}$$

$$R' = \begin{bmatrix} e'_1 & e'_2 & 0 \\ -e'_2 & e'_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C.105}$$

(f) Replace F by $R'FR^T$

The fundamental matrix now has a special form

$$F = \begin{pmatrix} ff'd & -f'c & -f'd \\ -fb & a & b \\ -fd & c & d \end{pmatrix} \tag{C.106}$$

(g) Set f = e_3,f' = e_3',a = F_{22},b = F_{23},c = F_{32} and d = F_{33}.

(h) The epipolar line in the first image passing through the epipole $(1, 0, f)^T$ and the image point $(0, t, 1)^T$ is the vector $(0, t, 1)^T$ X $(1, 0, f)^T = (tf, 1, -t)$

(i) The corresponding epipolar line is l'(t)=F$(0, t, 1)^T$

(j) The sum of the squared distances from each epipolar line to the origin is

$$s(t) = \frac{t^2}{1 + f^2t^2} + \frac{(ct + d)^2}{(at + b)^2 + f'^2(ct + d)^2} \tag{C.107}$$

(k) To find the minimum of this function the derivative is computed. Maxima and minima of s(t) will occur when s'(t) = 0 i.e at the roots of the below polynomial of degree six. Therefore it may have 6 real roots corresponding to 3 minima and 3 maxima of the function s(t). The absolute minima is found by evaluating s(t) at each of the roots of s'(t).

$$s'(t) = t((at + b)^2 + f'^2(ct + d)^2)^2 - (ad - bc)(1 + f^2t^2)^2(at + b)(ct + d) = 0 \tag{C.108}$$

If f and f' is set to 1 then if s(t)=0 when t=0 the corresponding points x and x' will exactly satisfy the epipolar constraint.

(l) Evaluate the two lines l(t)= $(tf, 1, -t)$ and l'(t)=F$(0, t, 1)^T$ at the minimum t just found. $\hat{x}$ and $\hat{x}'$ are the closet points on these lines to the origin. For a general line (λ, μ, v) The closest point on the line to the origin is $(-\lambda v, -\mu v, \lambda^2 + \mu^2)$.

(m) Transfer the found coordinates back to the original system by replacing $\hat{x}$ and $\hat{x}'$ by $T^{-1}R^T\hat{x}$ and $T^{-1}R'^T\hat{x}'$

(n) Determine the scene coordinate by the homogeneous method mentioned above.

C.4.3 Comparison Of Simple And Optimal Triangulation

A comparison of the two triangulation methods can be seen below in figure C.15. The first method uses the midpoint of the common perpendicular to the ray and the second method is the optimal method discussed in the previous section.

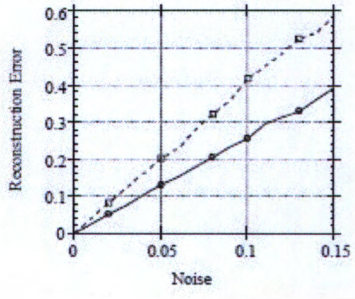


FIGURE C.15: The top line is the triangulation method which uses the midpoint of the common perpendicular to the rays (not projectively invariant) and the bottom line is the optimal method. On the horizontal axis is the noise in image coordinates and on the vertical axis is the triangulation error (Hartley and Zisserman, 2003).

C.4.4 Uncertainty Of Triangulation

The uncertainty of triangulation depends on the angle between camera rays. The smaller the angle the weaker the reconstruction and vice versa. This can be seen below in image C.16.

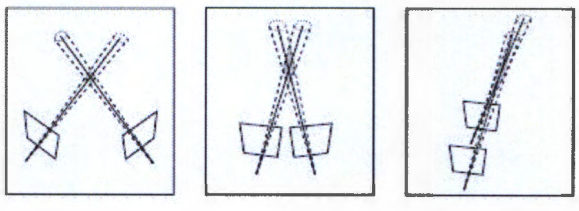


FIGURE C.16: The shaded region is the uncertainty region and it depends on the angle between the rays. The angle of rays determines the accuracy of the reconstruction. When the angle becomes smaller the points become less precisely localized and the reconstruction is weakened. Forward motion results in more parallel rays and thus weaker reconstructions (Hartley and Zisserman, 2003).

C.4.5 Types Of Reconstruction

In any reconstruction of point correspondences there is a true reconstruction of the actual points X and actual cameras P and P' that generated the measured observations. The reconstructed points and cameras differ from the true values by a transformation.

A reconstruction that is known to differ from the true reconstruction by a euclidean transformation is a euclidean reconstruction. A reconstruction that is known to differ from the true reconstruction by a similarity transformation is a similarity reconstruction and so forth.

Without knowing where a scene is placed it is impossible to reconstruct the absolute position or orientation of a scene. Therefore a scene is reconstructed at best to a euclidean reconstruction and therefore it is within a euclidean transformation (rotation and translation) of the world frame.

It is actually also impossible to determine the overall scale of a scene based on images. Therefore the scene is actually reconstructed at best to a similarity reconstruction and therefore it is within a similarity transformation (rotation,translation and scaling) of the true cameras and scene structure.

If the two cameras are uncalibrated yet are related via translational motion without change of calibration then an affine reconstruction is possible i.e the true reconstruction is within an affine transformation of the reconstruction.

If the cameras are calibrated the reconstruction must respect the angle between rays and so a similarity reconstruction is possible i.e the true reconstruction is within a similarity transformation of the reconstruction. This is because the scale cant be found.

If the cameras are not calibrated then the angle between rays will vary and a projective reconstruction will only be possible . To achieve a similarity reconstruction a stratified approach will need to be taken.

C.4.6 Stratified Reconstruction

The stratified approach to reconstruction involves beginning with a projective reconstruction and then refining it progressively to an affine and finally to a similarity reconstruction.

C.4.6.0.1 Projective To Affine Reconstruction

To progress to an affine reconstruction the plane at infinity π needs to be identified. Once the plane has been identified the transformation H that maps the plane to its true coordinates (0,0,0,1) is found. The transformation H is then applied to all points and the two cameras resulting in an affine reconstruction.

$$H^{-1}\pi = (0,0,0,1)^T \tag{C.109}$$

$$H = \begin{bmatrix} I|0 \\ \pi^T \end{bmatrix} \tag{C.110}$$

To determine the coordinates of the plane at infinity π three options may be followed:

1. Translational motion.

   If the camera undergoes pure translational motion a point X on the plane at infinity will map to the same point in two images related by a translation. With three such points the plane at infinity can be determined. As the motion is translational determining the fundamental matrix should involve a different approach.

2. Scene constraints.

   Scene constraints may be used. As long as three points can be identified that are known to lie on the plane at infinity the plane can be reconstructed. There are two ways to identify the plane at infinity.

   (a) Parallel lines.

   The intersection of two parallel lines in space gives a point on the plane at infinity. The image of this point is the vanishing point of the line and it is where the two images lines intersect. With three sets of parallel lines three distinct points can be found and the plane at infinity can be determined. Computing the intersection of lines in space is not a trivial task

(b) Distance ratios on a line.

This is an alternative to computing vanishing points as the intersection of parallel lines. The location of the vanishing points can be determined given two intervals on a line with a known length ratio. To determine the length ratio the world distance needs to be known.

Once the plane at infinity has been located, an image-to-image map called the 'infinite homography' has also been found. This map transfers points from the left image to the right image via the plane at infinity.

$$x' = H_\infty x \tag{C.111}$$

C.4.6.0.2 Affine To Similarity/Metric Reconstruction

The key to an affine reconstruction is the identification of the plane at infinity . The key to a similarity/metric reconstruction is the identification of the absolute conic Ω_∞

Once the absolute conic Ω_∞ has been identified the homography transformation H that maps the conic to its true equation on the plane is found. The transformation is then applied to all points and the two cameras resulting in a similarity reconstruction.

Instead of identifying the absolute conic it is easier to identify the image of the absolute conic ω. Once the image of the absolute conic has been identified the homography H which transforms the affine reconstruction to a metric reconstruction can be found.

Suppose that the image of the absolute conic which is seen by camera P is a conic ω

$$P = [M|m] \tag{C.112}$$

$$AA^T = \left(M^T wM\right)^{-1} \tag{C.113}$$

A is determined by Cholesky factorization of the above equation.

$$H = \begin{bmatrix} A^{-1} & 0 \\ 0 & 1 \end{bmatrix} \tag{C.114}$$

The image of the absolute conic ω can be determined using the following sources of constraints:

1. Constraints from scene orthogonality.

   Vanishing points are the locations in images where parallel lines intersects. Pairs of vanishing points v1 and v2 from orthogonal scene lines place a linear constraint on ω

   $$v_1^T \omega v_2 = 0 \tag{C.115}$$

2. Same camera used in both images.

   If the same camera is used in both images the calibration matrix will be the same, this means that the image of the absolute conic will be the same in both images as it depends solely on the calibration matrix and not on the position or orientation

of the camera i.e $\omega'= \omega$. Therefore since the absolute conic lies on the plane of infinity which was determined in the previous section, its image may be transfered from one view to another via the infinite homography and the following equation can be formulated.

$$\omega' = \omega = H_{\infty}^{-T} w H_{\infty}^{-1} \tag{C.116}$$

A set of linear equations on ω has been formed. In general this set places four constraints on ω .

As ω has 5 degrees of freedom by combining the linear equations from the above sources of constraints ω may be determined uniquely.

C.4.6.0.3 Projective To Similarity Reconstruction (Direct)

If the image of the absolute conic ω is known it is possible to compute the calibration matrix K and then directly achieve a similarity reconstruction.

$$\omega = K^{-T}K^{-1} \tag{C.117}$$

ω is simply inverted and then cholesky factorization is performed to obtain K. K can be determined for each camera like this . With calibrated cameras a similarity/metric reconstruction of the scene may be computed using the essential matrix.

If ω is not known it is possible to jump directly from a projective reconstruction to a true reconstruction if ground control points are known i.e the coordinates in the world frame are given. Then its possible to work out the homography between the true reconstruction and the 3D points . As H has 15 degrees of freedom and each point correspondence provides 3 equations a minimum of 5 points in needed.

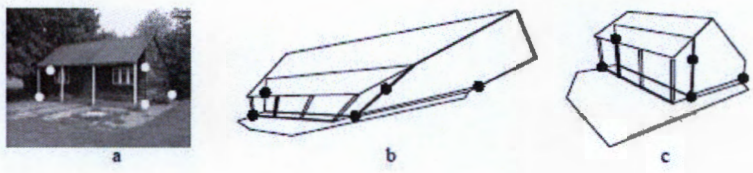


FIGURE C.17: Direct reconstruction: (a) Five world points (b) The corresponding points on the projective reconstruction (c) The reconstruction after the five points are mapped to their world positions (Hartley and Zisserman, 2003).

Appendix D

C++ Code

D.1 Main Class

tabsize

```
//Real-time Progressive Stereo Reconstruction
//RP3DR V10
//8 October
//Main class
//Matthew Westaway

//-------------------------------------------INCLUDES-------------------------------

//OpenCV include files
#include <opencv2\core\core.hpp>        //So we can use the basic types such as arrays i.e mat
#include <opencv2\highgui\highgui.hpp>  //So we can create windows to display images etc
#include <opencv2\calib3d\calib3d.hpp>  //So we can perform calibration and triangulation
#include <opencv2\xfeatures2d.hpp>

//PCL include files
#include <pcl\visualization\cloud_viewer.h>     //So we can visualize the 3D points

//Boost include files
#include <boost\lexical_cast.hpp>       //So we can return multiple variables from a function

//Intel thread building blocks include files
#include "tbb\tbb.h" //So our process and frame capture thread can run in parallel

//C++ Standard template library include files
#include <vector> //So we can store variables and randomly access them.
#include <map>  //So we can assign to each 3D point a map of in which frame it can be found
//and what index in that frame
#include <numeric>      //So we can determine the mean reprojection error

//C++ Standard library include files
#include <iostream>
#include <fstream>      //So we can save point cloud to a file

#include <math.h>
#include <time.h> //So we can time the processing

//Project include files
#include "opticalflow.h" //So we can perform optical flow matching - if required

//Namespace declarations
//So we can use the OpenCV library namespace without having to write cv before
using namespace cv;
//So we can use the C++ standard library namespace without having to write stl before
```

```
using namespace std;

//Initialize the amount of frames reconstructed
int frames_grabbed = 100000000;

//Declare the size of the frame
Size frame_size;

//Define PI to convert from Radians to Decimal Degrees
#define PI 3.14159265359;

// Launch the PCL point cloud viewer .
//We launch is here as it cannot be launched in a threaded function
pcl::visualization::CloudViewer viewer("Realtime Stereo Progressive Reconstruction");


//----------------TRUE DATA (CAMERA POSITIONS AND ORIENTATIONS IN WORLD SYSTEM)--

vector<vector<double>> camera_positions_true{
{ -8.31326, -6.3181, 0.16107 },
{ -9.46627, -5.58174, 0.147736 },
{ -10.8142, -4.53704, 0.122293 },
{ -12.404, -3.81315, 0.110559 },
{ -14.1604, -3.32084, 0.0862032 },
{ -15.8818, -3.15083, 0.0592619 },
{ -17.6302, -3.36186, 0.0325247 },
{ -19.6309, -3.81958, -0.00781603 },
{ -20.9553, -4.61897, -0.0303931 },
{ -21.9937, -5.82033, -0.0463931 } };

vector<vector<double>> camera_orientations_true{
{ 96.95663072   ,0.000148752 ,- 54.39273996},
{ 95.31112816 ,- 0.021335621 ,- 48.16937543},
{ 95.1066128,    0.00477105 ,- 37.32849172},
{ 92.84821072,   0.001584341 ,- 27.01884783},
{ 92.70045937 ,- 0.003447095, - 15.68810939},
{ 91.96382049 ,- 0.00117536, - 5.780346663},
{ 91.22832228,   0.00170578       ,5.415728389},
{ 91.43497451, - 0.008951083,    21.72369498},
{ 92.21319866, - 0.014586913     ,32.70540355},
{ 93.0069618 ,- 0.021576568,     44.96693268}};


double scale_distance = 1.628089953;//True dist btwn the first two cameras - used to scale reconstruction


vector<Mat> camera_positions_measured;  //Vector to store the measured camera positions
vector<Mat> camera_orientations_measured;//Vector to store the measured camera orientations
vector<double> camera_positions_error;  //Vector to store the camera positions errors
vector<Mat> camera_orientations_error;//Vector to store the camera orientations errors
vector<double> camera_orientations_x_error; //Vector to store the camera orientation errors wrt x
vector<double> camera_orientations_y_error; //Vector to store the camera orientation errors wrt y
vector<double> camera_orientations_z_error; //Vector to store the camera orientation errors wrt z
vector<double> projection_error; //Vector to store the mean reproj error for each frame
vector<double> projection_error_bound;//Vector to store the mean reproj error in the bound for each frame

double algorithmtime; //Computation time of algorithm

//Ratio of frame size to window size. To avoid displaying images too big for computer screen
double w = 4;

//Create a struct for each 3D point
struct CloudPoint
{
//We store the 3D point
Point3d pt;
//We store an index to the position of the right frame that reconstructed the point
//and an index to the position of the matched keypoint in that frame
multimap<int, int> frame_point;
//We store the reprojection error of that 3D point
double reprojection_error;
};
```

```
//-----------------------------------------CALIBRATE CAMERA FUNCTION-----------------------------------
//We pass the function the stream code its normally "1" for webcam
boost::tuple<Mat_<double>, Mat_<double> > Stream_Calibrate(int stream_code)
{
//We setup the input video stream
VideoCapture cap(stream_code);

//We check to see if the camera is connected
if (!cap.isOpened())
{
cout << "The camera is not connected. Connect the camera and re-enter the stream code" << endl;
int streamcode;
cin >> streamcode; //We get the new stream code from the user
Stream_Calibrate(streamcode); //Rerun function
}

//We get and set the width and height of the frames
frame_size.width = cap.get(CV_CAP_PROP_FRAME_WIDTH);
frame_size.height = cap.get(CV_CAP_PROP_FRAME_HEIGHT);

//We create window to show the stream
namedWindow("Stream_Calibrate", WINDOW_NORMAL);

//Resize the window to the defined display dimensions
resizeWindow("Stream_Calibrate", frame_size.width / w, frame_size.height / w);

//Ask the user if they would like a checkerbard to be displayed to the screen
cout << "Enter 'Y' if you would like a checkerboard to be displayed on the screen" << endl;
string calib_check;
cin >> calib_check;

if (calib_check == "Y") //If the user wants the calibration board on the screen we display it
{
//The location of the calibration pattern
Mat pattern = imread("checkerboard.png"); //Get checkerboard pattern


//If the calibration pattern cannot be found
if (!pattern.data)
{
cout << "The stream cannot be opened. Re-enter the stream code" << endl;
int streamcode;
cin >> streamcode;
Stream_Calibrate(streamcode);
}
Size checker = pattern.size(); //Determine the size of the calibration pattern

//Create a window to display the calibration pattern
namedWindow("Calibrate Pattern", WINDOW_NORMAL);

//Resize the window so that it is not too big
resizeWindow("Calibrate Pattern", checker.width / 3, checker.height / 3);
imshow("Calibrate Pattern", pattern); //Show the calibration pattern to the window
waitKey(30); //Its always important to have a waitKey so that the image has time to show.
}

Size patternsize(9, 6); //Dimensions of the checkerboard pattern

//We populate a vector of the known 54 object points i.e checkerboard corners

vector<Point3f> objectpoints;
for (int y = 0; y<6; ++y)
{
for (int x = 0; x<9; ++x)
{
objectpoints.push_back(Point3f(20 * x, 20 * y, 0));
}
}


Mat frame; //Frame to store the captured calibration pattern
//Vector for the image points in all views - 54 image points per view
vector<vector<Point2f>> imagepoints;
//Vector for the true 3D object points coordinates
//54 object points per view -all the same for each view
```

```
vector<vector<Point3f>> arrayObjectPoints;

//Initialise the counters
int calibration_no = 0;
int frame_no = 0;

//We begin a while loop that iterates until 10 calibration patterns have been captured
while (calibration_no <11)
{
//Check if it is possible to grab a frame and if so grab one.
bool bSuccess = cap.read(frame);
if (!bSuccess) //if not true
{
cout << "The stream cannot be accessed.  Re-enter the stream code" << endl;
int streamcode;
cin >> streamcode;
Stream_Calibrate(streamcode); //Rerun the function
}

//Show the frame
imshow("Stream_Calibrate", frame);
waitKey(30);

//Declare a new frame to convert the original frame to .
//As we want to convert the frame to grey
Mat frameGray;

cout << frame_no << endl; //Print out the frame number

//If its the 50th frame so grab the frame
if (frame_no % 50 == 0)
{
vector<Point2f> corners; //Vector for the 54 corner points

cvtColor(frame, frameGray, COLOR_BGR2GRAY); //Convert the frame to grey

//Check frame for the calibration pattern
bool found = findChessboardCorners(frameGray, patternsize, corners,
CV_CALIB_CB_ADAPTIVE_THRESH + CV_CALIB_CB_FAST_CHECK + CV_CALIB_CB_NORMALIZE_IMAGE);

//If calibration pattern is found
if (found)
{
//Print out the calibration frame no
cout << "Checkerboard no " << calibration_no << "of 10 detected" << endl;

//Get more accurate corners
cornerSubPix(frameGray, corners, Size(11, 11), Size(-1, -1),
TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

//Add the found corner points to the image point vector
imagepoints.push_back(corners);

//Add object points (we already know them) to the object point vector
arrayObjectPoints.push_back(objectpoints);

//Draw the corners
drawChessboardCorners(frame, patternsize, Mat(corners), found);

//Display corners captured briefly before going back to displaying the feed
imshow("Stream_Calibrate", frame);
waitKey(300);

//Clean the corners vector
corners.clear();
}

else
{
cout << "failed" << endl;        //If pattern not found
calibration_no--;        //Decrement the capture number so the capture will be reattemped
}

calibration_no++;                        //Increment the capture number
}
```

```
frame_no++; //Incremement the frame number
}

//Start calibration
Mat KMatrix; //Calibration matrix
Mat distCoeffs; //Distortion coefficients
//This is the rotation of the world (object points coordinate system)in the
//last image coordinate system.
vector<Mat> rvecs;
//This is the location of the world (object points coordinate system)in the
//last image coordinate system.
vector<Mat> tvecs;

//Calibrate the camera
double rms = calibrateCamera(arrayObjectPoints, imagepoints, frame_size,
KMatrix, distCoeffs, rvecs, tvecs);

cout << "the RMS is " << rms << endl;
cout << "the calibration matrix is " << KMatrix << endl;
cout << "the distortion matrix is " << distCoeffs << endl;

//Check to see if the user is happy with the calibration results and wants to begin reconstruction
string calib_status;

cout << "Enter key to commence 3D reconstruction (or enter RECALIBRATE)" << endl;
cin >> calib_status;
if (calib_status == "RECALIBRATE")
{
Stream_Calibrate(stream_code);
}

//Destroy the windows
destroyWindow("Stream_Calibrate");
destroyWindow("Calibrate Pattern");

//Return the calibration matrix and the distortion coefficients
return boost::make_tuple(KMatrix, distCoeffs);
}

//-----------------------------------READ IMAGES FROM FILE FUNCTION-----------------------------------
//We pass to the function the filename of the frames and the concurrent vector
//we want to store the frames to
void Filestream(string thefilename, tbb::concurrent_vector<Mat> &frames)

{
//We get the frames from the file
Mat frame = imread(thefilename.c_str());

//We check to see if the file exists
if (!frame.data)
{
cout << "The file address cannot be accessed. Enter the correct address" << endl;
string path;
cin >> path;
Filestream(path, frames);
}

//Get the size of the image
frame_size = frame.size();

//Create a window to display the frames
namedWindow("Filestream", WINDOW_NORMAL);
//Resize the window to half the file size
resizeWindow("Filestream", frame_size.width / w, frame_size.height / w);


//Initialize some variables
int frame_no = 1;

//Begin an infinite capturing loop to display all the images
while (1)
{
//Read frame
frame = imread(thefilename.c_str());
```

```
//If the frame has no data then break
if (!frame.data)
{
cout << "There is no data remaining" << endl;
break;
}

//Display the frame
imshow("Filestream", frame);
waitKey(30);

//Add the frame to the threaded processing vector
frames.push_back(frame);

//Increase the number of frames
frame_no = frame_no + 1;
cout << frame_no << endl;

//Get the new frame file name

if (frame_no<10)
{
string frame_Number = boost::lexical_cast<string, int>(frame_no);
thefilename = thefilename.substr(0, thefilename.size() - 5);
thefilename = thefilename.append(frame_Number);
thefilename = thefilename.append(".png");
}
else if (frame_no >= 10 && frame_no <= 100)
{
string frame_Number = boost::lexical_cast<string, int>(frame_no);
thefilename = thefilename.substr(0, thefilename.size() - 6);
thefilename = thefilename.append(frame_Number);
thefilename = thefilename.append(".png");

}
else if (frame_no>100 && frame_no <= 1000)
{
string frame_Number = boost::lexical_cast<string, int>(frame_no);
thefilename = thefilename.substr(0, thefilename.size() - 7);
thefilename = thefilename.append(frame_Number);
thefilename = thefilename.append(".png");

}
else if (frame_no >1000)
{
string frame_Number = boost::lexical_cast<string, int>(frame_no);
thefilename = thefilename.substr(0, thefilename.size() - 8);
thefilename = thefilename.append(frame_Number);
thefilename = thefilename.append(".png");

}

}

//Set the number of frames grabbed to the number of frames grabbed
frames_grabbed = frame_no - 1;
cout << "Frames grabbed : " << frames_grabbed << endl;

destroyWindow("Filestream");
}


//-----------------------------------------READ FRAMES FROM STREAM FUNCTION-----------------------------------------
//We get the width and height of the frames
void Livestream(int stream_code, tbb::concurrent_vector<Mat> &frames)
{

//Setup input video stream
VideoCapture cap(stream_code);//The code for the input video stream
//Set the width and height of the video frames to the incoming feed width.
frame_size.width = cap.get(CV_CAP_PROP_FRAME_WIDTH);
frame_size.height = cap.get(CV_CAP_PROP_FRAME_HEIGHT);

if (!cap.isOpened())
{
```

```
cout << "The camera is not connected. Connect the camera and re-enter the stream code" << endl;
int streamcode;
cin >> streamcode;
Livestream(streamcode, frames); //Rerun function
}

//Create window to show the stream
namedWindow("Livestream", WINDOW_NORMAL);
//Resize the window to the defined display height
resizeWindow("Livestream", frame_size.width / w, frame_size.height / w);


//Set up variables to save the frames to file
vector<int> compression_params; //Vector that stores the compression parameters of the image
compression_params.push_back(CV_IMWRITE_JPEG_QUALITY); //Specify the compression technique
compression_params.push_back(98); //Specify the compression quality

//Initialize the amount of frames passed and number of the frames captured
int frame_no = 1;
int frames_captured = 0;

//Begin an infinite capturing loop to stream the video input
while (1)
{
//Read a frame
Mat frame;
bool bSuccess = cap.read(frame);
if (!bSuccess) //Test to see if can read a frame
{
cout << "Cannot read frames anymore. Capturing is over . Processing will continue " << endl;
//Set the number of frames grabbed to the number of frames processed.
frames_grabbed = frames_captured;

break;
}

//Display the frame
imshow("Livestream", frame);
waitKey(30);

//Print out the frame_number
cout << frame_no << endl;

//If the frames are more than 100 frames grab a frame every 50 frames
if (frame_no>100 & frame_no % 50 == 0)
{
//Add the frame to the threaded processing vector
frames.push_back(frame);

//Print out the number of captured frames
cout << "Captured frame number " << frames_captured << endl;

//We write the frame to file
string name =  boost::lexical_cast<string, int>(frames_captured)+".jpg";
bool bSuccess = imwrite(name, frame, compression_params); //Write the frame to file
if (!bSuccess) //Check if can write the frame to file
{
cout << "ERROR : Failed to save the frame. Will restart the processing" << endl;
frames.clear(); //Clear the vector
Livestream(stream_code, frames);
}

//Incremement the number of frames captured
frames_captured++;

}

//Increment the frames passed
frame_no++;

//If the escape key is pressed for 30 seconds exit the infinite loop .
if (waitKey(30) == 27)
{
cout << "The escape key has been pressed. Capturing is over. Processing will continue" << endl;
```

```
//Set the number of frames reconstructed to the number of frames processed.
frames_grabbed = frames_captured;

break;
}


}

destroyWindow("Livestream");


}

//Function to process the frames
void Stream_process(Mat KMatrix, Mat distcoeff, tbb::concurrent_vector<Mat> &frames,
string keypoint_detection,string keypoint_matching,string ego_motion,time_t start)
{
//Sleep this thread untill at least a few frames have been pushed into the vector
boost::this_thread::sleep(boost::posix_time::milliseconds(10000));

//Create windows
namedWindow("Frame left", WINDOW_NORMAL);        //Window_normal allows resize
namedWindow("Frame right", WINDOW_NORMAL);
namedWindow("Matches", WINDOW_NORMAL);
//Resize the window to the defined display height
resizeWindow("Frame left", frame_size.width / w, frame_size.height / w);
//Resize the window to the defined display height
resizeWindow("Frame right", frame_size.width / w, frame_size.height / w);
//Resize the window to the defined display height
resizeWindow("Matches", 2 * frame_size.width / w, frame_size.height / w);


//Initialise variables
vector <CloudPoint> pcloud;                                  //The point cloud
vector<KeyPoint> keypoints_left;          //Keypoints left frame
vector<KeyPoint> keypoints_right;                  //Keypoints right frame
//Extrinsic camera parameters - rotation of world in camera and ptn of world in camera
Mat R, Rodrigues_R, t;

//Create visualisation point cloud to populate
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud;
cloud.reset(new pcl::PointCloud<pcl::PointXYZ>);

//Initialise the counter
int a = 1;

int waitkey = 30;                //If this is 0 then we will need to click enter
//on each image in order to proceed at each step


//Set the baseline state - to has not been reconstructed
string baseline_state = "Baseline has not been reconstructed";


//Create log file to store the results
string logfile = "LOG RP3DRv10 -" + string(keypoint_detection)
+ " - " + string(keypoint_matching) + " - " + string(ego_motion) + ".txt";
ofstream openfile;
openfile.open(logfile.c_str());
openfile << "LOG RP3DRv10  -" + string(keypoint_detection)
+ " - " + string(keypoint_matching) + " - " + string(ego_motion) + ".txt"<<endl;
cout << "LOG -" + string(keypoint_detection) + " - "
+ string(keypoint_matching) + " - " + string(ego_motion) + ".txt";

//Create reconstruction file to store the results
string reconfile = "RECONSTRUCTION RP3DRv10" + string(keypoint_detection)
+ " - " + string(keypoint_matching) + " - " + string(ego_motion) + ".txt";
ofstream openfile2;
openfile2.open(reconfile.c_str());
openfile2 << "RECONSTRUCTION RP3DRv10" + string(keypoint_detection)
+ " - " + string(keypoint_matching) + " - " + string(ego_motion) + ".txt" << endl;

//Go through all the frames that have been grabbed
while (a<frames_grabbed)
```

```
{
try //Need a try- catch in case no frames have been captured so far,due to real time
{
//Grab frame from vector
Mat frame_left_distorted = frames.at(a - 1);
Mat frame_right_distorted = frames.at(a);

//-------------------------------------STEP 1 FRAME UNDISTORTION----------

//Undistort the frames - remove lens distortion
Mat frame_left;
Mat frame_right;

undistort(frame_left_distorted, frame_left, KMatrix, distcoeff);
undistort(frame_right_distorted, frame_right, KMatrix, distcoeff);

//Display the frames
imshow("Frame left", frame_left);
waitKey(waitkey);
imshow("Frame right", frame_right);
waitKey(waitkey);

//If baseline pair of frames i.e the first two frames have not been reconstructed
if (baseline_state != "reconstructed")
{
//Set the initial camera extrinsic parameters to the true values
R = (Mat_<double>(3, 3) << 0.450927, -0.892535, 0.00679989,
-0.0945642, -0.0401974, 0.994707,
-0.887537, -0.449183, -0.102528
);
Rodrigues(R, Rodrigues_R);

t = (Mat_<double>(3, 1) << -3.48046703877, -1.19648323093, -9.84483520681);

Mat cameraextrinsic_left(3, 4, R.type());
cameraextrinsic_left(Range(0, 3), Range(0, 3)) = R * 1;
cameraextrinsic_left(Range(0, 3), Range(3, 4)) = t * 1;


//-------------------------------------STEP 1 KEYPOINT DETECTION ----------

//Start timer now for keypoint detection
time_t keypoint_timestart = time(0);

Ptr<Feature2D> f2d;

if (keypoint_detection == "SIFT")
{
f2d = xfeatures2d::SIFT::create();
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "SURF")
{
f2d = xfeatures2d::SURF::create();
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "BRISK")
{
f2d = BRISK::create();
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "AKAZE")
{
f2d = AKAZE::create();
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}
```

```
else if (keypoint_detection == "HOUGH")
{
//Detect and display hough line keypoints in the left frame
Mat canny_left, canny_left_colour;
Canny(frame_left, canny_left, 50, 200, 3); //Run canny edge detection
//Convert the canny image to colour
cvtColor(canny_left, canny_left_colour, CV_GRAY2BGR);

//Run hough lines on canny
vector<Vec4i> lines_left;
HoughLinesP(canny_left, lines_left, 1, CV_PI / 180, 50, 50, 10);

//Go through each hough line and populate keypoints
for (size_t i = 0; i < lines_left.size(); i++)
{
Vec4i l = lines_left[i];
//Draw the line on the coloured canny image
line(canny_left_colour, Point(l[0], l[1]),
Point(l[2], l[3]), Scalar(0, 0, 255), 3, CV_AA);

//choose only end points as keypoints
keypoints_left.push_back(KeyPoint(l[0], l[1], 1));
keypoints_left.push_back(KeyPoint(l[2], l[3], 1));

////Choose all points on the line to be keypoints
//LineIterator it(frame_left, Point(l[0], l[1]), Point(l[2], l[3]), 8);

//for (int t = 0; t < it.count; t++)
//{
//      keypoints_left.push_back(KeyPoint(it.pos().x, it.pos().y, 1));
//      it++;

//}

}

//Detect and display hough line keypoints in the right image
Mat canny_right, canny_right_colour;
Canny(frame_right, canny_right, 50, 200, 3); //Run canny edge detection
//Convert the canny image to colour
cvtColor(canny_right, canny_right_colour, CV_GRAY2BGR);


vector<Vec4i> lines_right;
HoughLinesP(canny_right, lines_right, 1, CV_PI / 180, 50, 50, 10);

//Go through each hough line and populate keypoints
for (size_t i = 0; i < lines_right.size(); i++)
{
Vec4i l = lines_right[i];
//Draw the line on the coloured canny image
line(canny_right_colour, Point(l[0], l[1]),
Point(l[2], l[3]), Scalar(0, 0, 255), 3, CV_AA);

//choose only end points as keypoints
keypoints_right.push_back(KeyPoint(l[0], l[1], 1));
keypoints_right.push_back(KeyPoint(l[2], l[3], 1));

//Choose all points on the line to be keypoints
//LineIterator it(frame_right, Point(l[0], l[1]), Point(l[2], l[3]), 8);

//for (int t = 0; t < it.count; t++)
//{
//      keypoints_right.push_back(KeyPoint(it.pos().x, it.pos().y, 1));
//      it++;
//
//}

}
}

//Display the keypoints
Mat keypoints_left_image;
drawKeypoints(frame_left, keypoints_left, keypoints_left_image);
imshow("Frame left", keypoints_left_image);
```

```
waitKey(waitkey);

Mat keypoints_right_image;
drawKeypoints(frame_right, keypoints_right, keypoints_right_image);
imshow("Frame right", keypoints_right_image);
waitKey(waitkey);

//Record the time as keypoints have now been computed
double keypoint_time = difftime(time(0), keypoint_timestart);

//------------------------------------STEP 2 KEYPOINT MATCHING ----------

//Start timer now for keypoint matching
time_t matching_detectionstart = time(0);

vector<DMatch> matches; //Matches

if (keypoint_matching == "opticalflow")
{
//Get matches using optical flow
Opticalflow view(frame_left, frame_right);
matches = view.Getmatches(keypoints_left, keypoints_right);
}

else if (keypoint_matching == "featurebrute")
{
//Get matches using descriptor feateure matching
Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);

//Match the descriptors
BFMatcher matcher(NORM_L2, true); //Use the BF matcher
matcher.match(descriptors_left, descriptors_right, matches);
}



else if (keypoint_matching == "featurebrutebinary")
{
//Get matches using descriptor feateure matching
Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);

//Match the descriptors
BFMatcher matcher(NORM_HAMMING, true); //Use the BF matcher
matcher.match(descriptors_left, descriptors_right, matches);
}

else if (keypoint_matching == "featureapprox")
{
Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);

//Match the descriptors
FlannBasedMatcher matcher; //Use th FLANN matcher
matcher.match(descriptors_left, descriptors_right, matches);

}

//Record the size of matches
int size_of_matches = matches.size();

//Draw the matches
Mat matches_image;
drawMatches(frame_left, keypoints_left, frame_right,
keypoints_right, matches, matches_image);
imshow("Matches", matches_image);
waitKey(waitkey);

//Convert the matched keypoints into alligned image points
vector<Point2f> alligned_points_left;
//They must be float as essential matrix and epipolar lines only take floats
```

```
vector<Point2f> alligned_points_right;
for (int i = 0; i < size_of_matches; i++)
{
alligned_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
alligned_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
}
//Record the time as matching has now been computed
double matching_time = difftime(time(0), matching_detectionstart);

//-------------------------------------STEP 3 EGO-MOTION ESTIMATION ----------

//Begin timing of ego-motion computation
time_t ego_motion_timestart = time(0);

Mat mask; //inliers
double focal = KMatrix.at<double>(0, 0);          //Focal distance
Point2d pp(KMatrix.at<double>(0, 2), KMatrix.at<double>(1, 2)); //Principal point
Mat essentialmatrix;
Mat fundamentalmatrix;
//We are going to triangulate these points so they must be double type not float
vector<Point2d> inlier_points_left;
vector<Point2d> inlier_points_right;
vector<DMatch> refined_matches;
int size_of_refinedmatches;


if (ego_motion == "5point")
{
essentialmatrix = findEssentialMat(alligned_points_left, alligned_points_right,
focal, pp, RANSAC, 0.999, 1, mask); //Use RANSAC for 5-point Essential Matrix
fundamentalmatrix = (KMatrix.t()).inv()*essentialmatrix*(KMatrix.inv());

//Get and display the inlier points from the essential matrix
for (int i = 0; i<size_of_matches; i++)
{
if (0 + mask.at<uchar>(i, 0) == 1)
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}
size_of_refinedmatches = refined_matches.size(); //Size of refined matches
}

else if (ego_motion == "7point")
{

vector<uchar> status(alligned_points_left.size());
//Calculate the fundamental matrix using ransac
Mat F_RANSAC = findFundamentalMat(alligned_points_left,
alligned_points_right, FM_RANSAC, 1, 0.999, status);


//Get and display the inlier points from the essential matrix
for (int i = 0; i < size_of_matches; i++)
{
if (status[i])
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}

fundamentalmatrix = findFundamentalMat(inlier_points_left,
inlier_points_right, FM_7POINT);  //Calculate the fundamental matrix using 7 - point
essentialmatrix = KMatrix.t()*fundamentalmatrix*KMatrix;

size_of_refinedmatches = refined_matches.size(); //Size of refined matches

}

else if (ego_motion == "8point")
{
```

```
vector<uchar> status(alligned_points_left.size());

Mat F_RANSAC = findFundamentalMat(alligned_points_left, alligned_points_right,
FM_RANSAC, 1, 0.999, status); //Calculate the fundamental matrix using ransac

//Get and display the inlier points from the essential matrix
for (int i = 0; i < size_of_matches; i++)
{
if (status[i])
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}
//Calculate the fundamental matrix using 8 - point algorithm
fundamentalmatrix = findFundamentalMat(inlier_points_left,
inlier_points_right, FM_8POINT);

essentialmatrix = KMatrix.t()*fundamentalmatrix*KMatrix;
size_of_refinedmatches = refined_matches.size(); //Size of refined matches

}
//Record the time as ego_motion has now been computed
double ego_motion_time = difftime(time(0), ego_motion_timestart);

//Draw the refined matches
Mat matches_refined_image;
drawMatches(frame_left, keypoints_left, frame_right,
keypoints_right, refined_matches, matches_refined_image);
imshow("Matches", matches_refined_image);
waitKey(waitkey);

//Draw the epipolar lines
vector<Vec3f> epipolarlines_left;
computeCorrespondEpilines(alligned_points_left, 1,
fundamentalmatrix, epipolarlines_left);

for (vector<Vec3f>::const_iterator it = epipolarlines_left.begin();
it != epipolarlines_left.end(); ++it)
{
line(frame_left, Point(0, -(*it)[2] / (*it)[1]),
Point(frame_left.cols, -((*it)[2] + (*it)[0] * frame_left.cols) / (*it)[1]),
Scalar(0, 0, 0));
}
imshow("Frame left", frame_left);
waitKey(waitkey);

vector<Vec3f> epipolarlines_right;
computeCorrespondEpilines(alligned_points_right, 2,
fundamentalmatrix, epipolarlines_right);

for (vector<Vec3f>::const_iterator it = epipolarlines_right.begin();
it != epipolarlines_right.end(); ++it)
{
line(frame_right, Point(0, -(*it)[2] / (*it)[1]),
Point(frame_right.cols, -((*it)[2] + (*it)[0] * frame_right.cols) / (*it)[1]),
Scalar(0, 0, 0));
}
imshow("Frame right", frame_right);
waitKey(waitkey);

//Recover the relative translation and rotation between frames
Mat R_between, Rodrigues_R_between, t_between;
recoverPose(essentialmatrix, alligned_points_left, alligned_points_right,
R_between, t_between, focal, pp, mask);
Rodrigues(R_between, Rodrigues_R_between);

//Define the second camera extrinsic matrix (i.e normalized camera matrix)
composeRT(Rodrigues_R, t, Rodrigues_R_between, scale_distance*t_between, Rodrigues_R, t);
Rodrigues(Rodrigues_R, R);

//Recover the camera orientaton and position
Mat camera_orientation = R.t();
cout << camera_orientation << endl;
```

```
double Rx = atan2(camera_orientation.at<double>(2, 1) ,
camera_orientation.at<double>(2, 2)) * 180 / PI;
double Ry = atan2(-camera_orientation.at<double>(2, 0),
sqrt(pow(camera_orientation.at<double>(2, 1),2) +
pow(camera_orientation.at<double>(2, 2),2))) * 180 / PI;
double Rz = atan2(camera_orientation.at<double>(1, 0) ,
camera_orientation.at<double>(0, 0)) * 180 / PI;
Mat camera_euler_orientation = (Mat_<double>(3, 1) << Rx, Ry, Rz);
Mat camera_position = -R.t()*t;

//Add the measured cameras position ,orientation to the global vectors
camera_positions_measured.push_back(camera_position);
camera_orientations_measured.push_back(camera_euler_orientation);

//Add the measured cameras position ,orientation error to the global vectors
double positionerror = cv::norm(camera_position, Mat(camera_positions_true[a - 1]));
Mat orientationerror = abs(camera_euler_orientation - Mat(camera_orientations_true[a - 1]));

camera_positions_error.push_back(positionerror);
camera_orientations_error.push_back(orientationerror);
camera_orientations_x_error.push_back(orientationerror.at<double>(0));
camera_orientations_y_error.push_back(orientationerror.at<double>(1));
camera_orientations_z_error.push_back(orientationerror.at<double>(2));

//Set extrinsic paramaters i.e normalized camera matrix
Mat cameraextrinsic_right(3, 4, R.type()); // T is 4x4
cameraextrinsic_right(Range(0, 3), cv::Range(0, 3)) = R * 1; // copies R into T
cameraextrinsic_right(Range(0, 3), cv::Range(3, 4)) = t * 1;

//-------------------------------STEP 4 RECONSTRUCTION (OPTIMUM METHOD) ----------

//Prepare the inlier points for triangulation
Mat inlier_points_left_mat = Mat(inlier_points_left); //Convert them to Mat
Mat inlier_points_right_mat = Mat(inlier_points_right);

//Prepare the inlier points for correct matches function must be 2 channel 1 row
Mat inlier_points_left_mat_reshape = inlier_points_left_mat.reshape(2, 1);
Mat inlier_points_right_mat_reshape = inlier_points_right_mat.reshape(2, 1);

//Correct matches
correctMatches(fundamentalmatrix, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape);

//Set bounds inner 90% of image
double x_upper = 0.9*frame_size.width;
double x_lower = 0.1*frame_size.width;
double y_upper = 0.9*frame_size.height;
double y_lower = 0.1*frame_size.height;

//Get the projecton matrices
Mat projection_left = KMatrix*cameraextrinsic_left;
Mat projection_right = KMatrix*cameraextrinsic_right;

//Triangulate the points
Mat points4D;
triangulatePoints(projection_left, projection_right, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape, points4D);

//The triangulated points are homogeneous so we must convert them to normal
Mat points3D;
Mat x = Mat(points4D).t();
convertPointsFromHomogeneous(x, points3D);

//Reproject the triangulated points to the image and determine the reprojection error
Mat reprojected_points;
projectPoints(points3D, Rodrigues_R, t, KMatrix, Mat(), reprojected_points);
//The distortion matrix is empty Mat() as the images have already been undistorted

//Vector to store reprojection errors and reprojection errors in bound
vector<double> reproject_error;
vector<double> reproject_error_bound;

int in_bound = 0; //Number of points within bound
```

```
//Go through each refined matching point. i.e point that has been reconstructed
for (int i = 0; i<size_of_refinedmatches; i++)
{
//Get the image point
Mat right = (Mat_<double>(2, 1) << inlier_points_right_mat.at<double>(i, 0),
inlier_points_right_mat.at<double>(i, 1)); //The original image point
Mat right_reprjected = (Mat_<double>(2, 1) << reprojected_points.at<double>(i, 0),
reprojected_points.at<double>(i, 1)); //The reprojected image point

//Calculate the reprojection error
double re_error = norm(right - right_reprjected); //Reprojection error

//Check if points are within bound
if (inlier_points_left_mat.at<double>(i, 0) < x_upper &&
inlier_points_left_mat.at<double>(i, 0) > x_lower &&
inlier_points_left_mat.at<double>(i, 1) < y_upper &&
inlier_points_left_mat.at<double>(i, 1) > y_lower &&
inlier_points_right_mat.at<double>(i, 0) < x_upper &&
inlier_points_right_mat.at<double>(i, 0) > x_lower &&
inlier_points_right_mat.at<double>(i, 1) < y_upper &&
inlier_points_right_mat.at<double>(i, 1) > y_lower)
{
//Add 3D point to our global point cloud
CloudPoint newpoint;
newpoint.pt = Point3d(points3D.at<double>(i, 0), points3D.at<double>(i, 1),
points3D.at<double>(i, 2));
//Insert the position of the right frame and the position of the image point in the right frame
newpoint.frame_point.insert(pair<int, int>(a, refined_matches[i].trainIdx));
//Insert the reprojection error
newpoint.reprojection_error = re_error;
pcloud.push_back(newpoint);

//Add 3D point to pcl visualizer
pcl::PointXYZ pclp; // Convert 3D point type to PCL type
pclp.x = points3D.at<double>(i, 0);
pclp.y = points3D.at<double>(i, 1);
pclp.z = points3D.at<double>(i, 2);
cloud->push_back(pclp);

//Push back the reprojection error
reproject_error_bound.push_back(re_error);

//Increase the number of points in the bound
in_bound++;
}

//Push back the reprojection error - as if every point was triangulated
reproject_error.push_back(re_error);

}

//Record the computation time so far
double recon_time = difftime(time(0), start);

//Work out the mean and stdev reprojection error
double sum_reprojection = accumulate(begin(reproject_error), end(reproject_error), 0.0);
double mean_reprojection = sum_reprojection / reproject_error.size();

double accum_reprojection = 0.0;
for_each(begin(reproject_error), end(reproject_error), [&](const double d)
{
accum_reprojection += (d - mean_reprojection) * (d - mean_reprojection); //functor
}
);

double stdev_reprojection = sqrt(accum_reprojection / (reproject_error.size() ));

//Work out the mean and stdev reprojection error in the bound

double sum_reprojection_bound = accumulate(begin(reproject_error_bound),
std::end(reproject_error_bound), 0.0);
double mean_reprojection_bound = sum_reprojection_bound / reproject_error_bound.size();

double accum_reprojection_bound = 0.0;
```

```
for_each(begin(reproject_error_bound), end(reproject_error_bound), [&](const double d)
{
accum_reprojection_bound += (d - mean_reprojection_bound) *
(d - mean_reprojection_bound); //functor
}
);

double stdev_reprojection_bound = sqrt(accum_reprojection_bound /
(reproject_error_bound.size() ));

//Add the mean reprojection error to the reprojection error
//vector of all mean reprojection errors
projection_error.push_back(mean_reprojection);

//Add the mean reprojection error in the bound to the
//reprojection error vector of all mean reprojection bound errors
projection_error_bound.push_back(mean_reprojection_bound);

//Set the state of the baseline to constructed now
baseline_state = "reconstructed";

//Show the cloud
viewer.showCloud(cloud);

//Output to screen
cout << "_____" << endl;
cout << "_____" << endl;
cout << "Frame number: " << a << endl;
cout << " " << endl;
cout << "Number of keypoints: " << keypoints_left.size() << " "
<< keypoints_right.size() << endl;
cout << " " << endl;
cout << "Number of matches: " << size_of_matches << " " << endl;
cout << " " << endl;
cout << "Number of reconstruction points (refined matches): "
<< size_of_refinedmatches << endl;
cout << " " << endl;
cout << "Number of reconstruction points (within bound ): "
<< in_bound << endl;
cout << " " << endl;
cout << "Reprojection error : " << mean_reprojection << " "
<< stdev_reprojection << endl;
cout << " " << endl;
cout << "Reprojection error in bound: " << mean_reprojection_bound
<< " " << stdev_reprojection_bound << endl;
cout << " " << endl;
cout << "Point cloud size: " << pcloud.size() << endl;
cout << " " << endl;
cout << "Normalized camera matrix (Extrinsic parameters) " << endl;
cout << cameraextrinsic_right << endl;
cout << " " << endl;
cout << "Camera position: " << endl;
cout << camera_position << endl;
cout << " " << endl;
cout << "Camera orientation    " << endl;
cout << camera_orientation << endl;
cout << " " << endl;
cout << "Camera orientation (euler XYZ)  " << endl;
cout << camera_euler_orientation << endl;
cout << " " << endl;
cout << "Time of keypoint detection, matching , ego-motion estimation" <<
keypoint_time << " " << matching_time << " " << ego_motion_time << endl;
cout << " " << endl;
cout << "Time of frame reconstruction " << recon_time << endl;
cout << " " << endl;
cout << "_____" << endl;
cout << "_____" << endl;

//Output to log file
openfile << "_____" << endl;
openfile << "_____" << endl;
openfile << "Frame number: " << a << endl;
openfile << " " << endl;
openfile << "Number of keypoints: " << keypoints_left.size() << " "
<< keypoints_right.size() << endl;
```

```
openfile << " " << endl;
openfile << "Number of matches: " << size_of_matches << " " << endl;
openfile << " " << endl;
openfile << "Number of reconstruction points (refined matches): "
<< size_of_refinedmatches << endl;
openfile << " " << endl;
openfile << "Number of reconstruction points (within bound ): "
<< in_bound << endl;
openfile << " " << endl;
openfile << "Reprojection error : " << mean_reprojection << " "
<< stdev_reprojection << endl;
openfile << " " << endl;
openfile << "Reprojection error in bound: " << mean_reprojection_bound << " "
<< stdev_reprojection_bound << endl;
openfile << " " << endl;
openfile << "Point cloud size: " << pcloud.size() << endl;
openfile << " " << endl;
openfile << "Normalized camera matrix (Extrinsic parameters) " << endl;
openfile << cameraextrinsic_right << endl;
openfile << " " << endl;
openfile << "Camera position: " << endl;
openfile << camera_position << endl;
openfile << " " << endl;
openfile << "Camera orientation " << endl;
openfile << camera_orientation << endl;
openfile << " " << endl;
openfile << "Camera orientation (euler angles)" << endl;
openfile << camera_euler_orientation.t() << endl;
openfile << " " << endl;
openfile << "Time of keypoint detection , matching , ego-motion estimation"
<< keypoint_time << " " << matching_time << " " << ego_motion_time << endl;
openfile << " " << endl;
openfile << "Time of frame reconstruction " << recon_time << endl;
openfile << " " << endl;
openfile << "-------------------------------------------" << endl;
openfile << "-------------------------------------------" << endl;


}

//If baseline pair of frames has been reconstructed
else if (baseline_state == "reconstructed")
{
//Begin timing of this frame
time_t frame_time = time(0);

//Set the previous pair of frames right camera extrinsic matrix
//to this new pairs left camera extrinsic matrix
Mat cameraextrinsic_left(3, 4, R.type());
cameraextrinsic_left(Range(0, 3), Range(0, 3)) = R * 1;
cameraextrinsic_left(Range(0, 3), Range(3, 4)) = t * 1;

//Set the previous pair of frames right keypoints to this new pairs left camera keypoints.
keypoints_left = keypoints_right;
keypoints_right.clear(); //clear right keypoints

//------------------------------------STEP 1 KEYPOINT DETECTION ------------

Ptr<Feature2D> f2d;

if (keypoint_detection == "SIFT")
{
f2d = xfeatures2d::SIFT::create();
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "SURF")
{
f2d = xfeatures2d::SURF::create();
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "BRISK")
{
f2d = BRISK::create();
```

```
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "AKAZE")
{
f2d = AKAZE::create();
f2d->detect(frame_left, keypoints_left);
f2d->detect(frame_right, keypoints_right);
}

else if (keypoint_detection == "HOUGH")
{
//Detect and display hough line keypoints in the right image
Mat canny_right, canny_right_colour;
Canny(frame_right, canny_right, 50, 200, 3); //Run canny edge detection
cvtColor(canny_right, canny_right_colour, CV_GRAY2BGR); //Convert the canny image to colour

vector<Vec4i> lines_right;
HoughLinesP(canny_right, lines_right, 1, CV_PI / 180, 50, 50, 10);

//Go through each hough line and populate keypoints
for (size_t i = 0; i < lines_right.size(); i++)
{
Vec4i l = lines_right[i];
line(canny_right_colour, Point(l[0], l[1]), Point(l[2], l[3]),
Scalar(0, 0, 255), 3, CV_AA);

//choose just end points to be keypoints
keypoints_right.push_back(KeyPoint(l[0], l[1], 1));
keypoints_right.push_back(KeyPoint(l[2], l[3], 1));

//Choose end points on the line to be keypoints
//LineIterator it(frame_right, Point(l[0], l[1]), Point(l[2], l[3]), 8);

//for (int t = 0; t < it.count; t++)
//{
//      keypoints_right.push_back(KeyPoint(it.pos().x, it.pos().y, 1));
//      it++;
//
//}

}
}


//Display the keypoints
Mat keypoints_left_image;
drawKeypoints(frame_left, keypoints_left, keypoints_left_image);
imshow("Frame left", keypoints_left_image);
waitKey(waitkey);

Mat keypoints_right_image;
drawKeypoints(frame_right, keypoints_right, keypoints_right_image);
imshow("Frame right", keypoints_right_image);
waitKey(waitkey);

//------------------------------------------STEP 2 KEYPOINT MATCHING ----------------------

vector<DMatch> matches; //Matches

if (keypoint_matching == "opticalflow")
{
//Get matches using optical flow
Opticalflow view(frame_left, frame_right);
matches = view.Getmatches(keypoints_left, keypoints_right);
}

else if (keypoint_matching == "featurebrute")
{
Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);
//Match the descriptors
BFMatcher matcher(NORM_L2, true); //Use the BF matcher
```

```
matcher.match(descriptors_left, descriptors_right, matches);

}

//Need to use binary BF Matcher if using AKAZE or BRISK
else if (keypoint_matching == "featurebrutebinary")
{
Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);
//Match the descriptors
BFMatcher matcher(NORM_HAMMING, true); //Use the BF matcher
matcher.match(descriptors_left, descriptors_right, matches);

}

else if (keypoint_matching == "featureapprox")
{

Mat descriptors_left, descriptors_right; //Descriptors
f2d->compute(frame_left, keypoints_left, descriptors_left);
f2d->compute(frame_right, keypoints_right, descriptors_right);

//Match the descriptors
FlannBasedMatcher matcher; //Use th FLANN matcher
matcher.match(descriptors_left, descriptors_right, matches);

}

//Record the size of matches
int size_of_matches = matches.size();

Mat matches_image;
drawMatches(frame_left, keypoints_left, frame_right,
keypoints_right, matches, matches_image);
imshow("Matches", matches_image);
waitKey(waitkey);

//Convert the matched keypoints into alligned image points
//They must be float as essential matrix and epipolar lines only take floats
vector<Point2f> alligned_points_left;
vector<Point2f> alligned_points_right;
for (int i = 0; i < size_of_matches; i++)
{
alligned_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
alligned_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
}

//-------------------------------------STEP 3 EGO-MOTION ESTIMATION ----------------

Mat mask; //inliers
double focal = KMatrix.at<double>(0, 0);                    //Focal distance
Point2d pp(KMatrix.at<double>(0, 2), KMatrix.at<double>(1, 2));    //Principal point
Mat essentialmatrix;
Mat fundamentalmatrix;
vector<Point2d> inlier_points_left;
//We are going to triangulate these points so they must be double type not float
vector<Point2d> inlier_points_right;
vector<DMatch> refined_matches;
int size_of_refinedmatches;


if (ego_motion == "5point")
{
//Use RANSAC for 5-point Essential Matrix
essentialmatrix = findEssentialMat(alligned_points_left,
alligned_points_right, focal, pp, RANSAC, 0.999,1, mask);

fundamentalmatrix = (KMatrix.t()).inv()*essentialmatrix*(KMatrix.inv());

//Get and display the inlier points from the essential matrix

for (int i = 0; i<size_of_matches; i++)
{
if (0 + mask.at<uchar>(i, 0) == 1)
```

```
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}
size_of_refinedmatches = refined_matches.size(); //Size of refined matches
}

else if (ego_motion == "7point")
{
vector<uchar> status(alligned_points_left.size());

//Calculate the fundamental matrix using ransac
Mat F_RANSAC = findFundamentalMat(alligned_points_left,
alligned_points_right, FM_RANSAC, 1, 0.999, status);


//Get and display the inlier points from the essential matrix

for (int i = 0; i < size_of_matches; i++)
{
if (status[i])
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}
//Calculate the fundamental matrix using 7 - point
fundamentalmatrix = findFundamentalMat(inlier_points_left,
inlier_points_right, FM_7POINT);

essentialmatrix = KMatrix.t()*fundamentalmatrix*KMatrix;

size_of_refinedmatches = refined_matches.size(); //Size of refined matches

}

else if (ego_motion == "8point")
{
vector<uchar> status(alligned_points_left.size());
//Calculate the fundamental matrix using ransac
Mat F_RANSAC = findFundamentalMat(alligned_points_left,
alligned_points_right, FM_RANSAC, 1, 0.999, status);


//Get and display the inlier points from the essential matrix

for (int i = 0; i < size_of_matches; i++)
{
if (status[i])
{
inlier_points_left.push_back(keypoints_left[matches[i].queryIdx].pt);
inlier_points_right.push_back(keypoints_right[matches[i].trainIdx].pt);
refined_matches.push_back(matches[i]);
}
}
//Calculate the fundamental matrix using 7 - point
fundamentalmatrix = findFundamentalMat(inlier_points_left,
inlier_points_right, FM_8POINT);
essentialmatrix = KMatrix.t()*fundamentalmatrix*KMatrix;

size_of_refinedmatches = refined_matches.size(); //Size of refined matches

}


//Draw the refined matches
Mat matches_refined_image;
drawMatches(frame_left, keypoints_left, frame_right,
keypoints_right, refined_matches, matches_refined_image);
imshow("Matches", matches_refined_image);
waitKey(waitkey);
```

```
//Draw the epipolar lines
vector<Vec3f> epipolarlines_left;
computeCorrespondEpilines(alligned_points_left , 1, fundamentalmatrix, epipolarlines_left);

for (vector<Vec3f>::const_iterator it = epipolarlines_left.begin();
it != epipolarlines_left.end(); ++it)
{
line(frame_left, Point(0, -(*it)[2] / (*it)[1]),
Point(frame_left.cols, -((*it)[2] + (*it)[0] * frame_left.cols) / (*it)[1]),
Scalar(0, 0, 0));
}
imshow("Frame left", frame_left);
waitKey(waitkey);

vector<Vec3f> epipolarlines_right;
computeCorrespondEpilines(alligned_points_right , 2,
fundamentalmatrix, epipolarlines_right);

for (vector<Vec3f>::const_iterator it = epipolarlines_right.begin();
it != epipolarlines_right.end(); ++it)
{
line(frame_right, Point(0, -(*it)[2] / (*it)[1]),
Point(frame_right.cols, -((*it)[2] + (*it)[0] * frame_right.cols) / (*it)[1]),
Scalar(0, 0, 0));
}
imshow("Frame right", frame_right);
waitKey(waitkey);


//Go through each match and see if it has already been reconstructed,
//if so record the 3D-2D correspondances

vector<Point2f> imgpoints;                          //Frame points in right frame
vector<Point3d> ppcloud;                      //Corresponding 3D points
vector<int> reconstructed;                          //Vector to prevent duplicates

for (int c = 0; c < size_of_refinedmatches; c++)
{
//The keypoint position in the previous frame
int index_in_old_view = refined_matches[c].queryIdx;

//Go through each 3D point
for (int d = 0; d < pcloud.size(); d++)
{
//Look to see if that points training index(right frame keypoint position)
//is equal to the previous frames keypoint

map<int, int>::iterator itr = pcloud[d].frame_point.find(a - 1);

if (itr != pcloud[d].frame_point.end())
{

if (itr->second == index_in_old_view)
{
//Add the 3D and 2D points for PNP
ppcloud.push_back(pcloud[d].pt);
Point2f pt = keypoints_right[refined_matches[c].trainIdx].pt;
imgpoints.push_back(pt);

//Record that this 3D point has already been reconstructed
reconstructed.push_back(c);

//Include the current frame and match training index position
//in the existing 3D points map.
//This is as we want to use the same points in future and
//not re-reconstruct them , this will avoid accumulated errors
pcloud[d].frame_point.insert(pair<int, int>(a, refined_matches[c].trainIdx));
break;
}
}
}
}

//Check to see that there are more than or equal to 4 points as PNP needs 4 points minimum
if (ppcloud.size() < 4)
```

```
{
cout << "There are only " << ppcloud.size()
<< "  3D-2D correspondances. 4 is the minimum , please try a different keypoint detector" << endl;
break;
}

Mat inliers;

//Recover the relative translation and rotation between frames
Mat R_between, Rodrigues_R_between, t_between;
recoverPose(essentialmatrix, alligned_points_left, alligned_points_right,
R_between, t_between, focal, pp, mask);
Rodrigues(R_between, Rodrigues_R_between);

//Determine the provisional extrinsic parameters of the second camera
composeRT(Rodrigues_R, t, Rodrigues_R_between,
scale_distance*t_between, Rodrigues_R, t);
Rodrigues(Rodrigues_R, R);

//Deteremine the camera extrinsic parameters using PNP Ransac.
solvePnPRansac(ppcloud, imgpoints, KMatrix, Mat(), Rodrigues_R, t, true,
500, 1, 0.999, inliers, SOLVEPNP_ITERATIVE);
Rodrigues(Rodrigues_R, R);

Mat cameraextrinsic_right(3, 4, R.type());
cameraextrinsic_right(Range(0, 3), Range(0, 3)) = R * 1;
cameraextrinsic_right(Range(0, 3), Range(3, 4)) = t * 1;

//Recover the camera orientaton and position
Mat camera_orientation = R.t();
double Rx = atan2(camera_orientation.at<double>(2, 1),
camera_orientation.at<double>(2, 2)) * 180/PI;
double Ry = atan2(-camera_orientation.at<double>(2, 0),
sqrt(pow(camera_orientation.at<double>(2, 1), 2) +
pow(camera_orientation.at<double>(2, 2), 2))) * 180 /PI ;
double Rz = atan2(camera_orientation.at<double>(1, 0),
camera_orientation.at<double>(0, 0)) * 180 / PI;
Mat camera_euler_orientation = (Mat_<double>(3, 1) << Rx, Ry, Rz);
Mat camera_position = -R.t()*t;

//Add the measured cameras position ,orientation to the global vectors
camera_positions_measured.push_back(camera_position);
camera_orientations_measured.push_back(camera_euler_orientation);

//Add the measured cameras position ,orientation error to the global vectors
Mat orientationerror = abs(camera_euler_orientation - Mat(camera_orientations_true[a - 1]));
double positionerror = cv::norm(camera_position, Mat(camera_positions_true[a - 1]));

camera_positions_error.push_back(positionerror);
camera_orientations_error.push_back(orientationerror);

camera_orientations_x_error.push_back(orientationerror.at<double>(0));
camera_orientations_y_error.push_back(orientationerror.at<double>(1));
camera_orientations_z_error.push_back(orientationerror.at<double>(2));

//-------------------------------------STEP 4 RECONSTRUCTION (OPTIMUM METHOD) ----------------

//Prepare the inlier points for correct matches
Mat inlier_points_left_mat = Mat(inlier_points_left);
Mat inlier_points_right_mat = Mat(inlier_points_right);
Mat inlier_points_left_mat_reshape = inlier_points_left_mat.reshape(2, 1);
Mat inlier_points_right_mat_reshape = inlier_points_right_mat.reshape(2, 1);

//Get min and max keypoints and set bounds . Inside 90% of image
double x_upper = 0.9*frame_size.width;
double x_lower = 0.1*frame_size.width;
double y_upper = 0.9*frame_size.height;
double y_lower = 0.1*frame_size.height;

//Correct matches
correctMatches(fundamentalmatrix, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape);

//Get the projecton matrices
```

```
Mat projection_left = KMatrix*cameraextrinsic_left;
Mat projection_right = KMatrix*cameraextrinsic_right;

//Triangulate the points
Mat points4D;
triangulatePoints(projection_left, projection_right, inlier_points_left_mat_reshape,
inlier_points_right_mat_reshape, points4D);

//Convert the triangulated points from homogeneous coordinates to normal coordinates.
Mat points3D;
Mat x = Mat(points4D).t();
convertPointsFromHomogeneous(x, points3D);

//Reproject the triangulated points to the image and determine the reprojection error
Mat reprojected_points;
projectPoints(points3D, Rodrigues_R, t, KMatrix, Mat(), reprojected_points);
//We use an empty Mat() for distortion coefficients
//as the image has already been undistorted.

vector<double> reproject_error;
vector<double> reproject_error_bound;
int in_bound = 0;

//Go through each matching point
for (int i = 0; i<size_of_refinedmatches; i++)
{

//If point has not already been reconstructed reconstruct point
if (std::find(reconstructed.begin(), reconstructed.end(), i) != reconstructed.end())
{
//already has been reconstructed
}
else
{
Mat right = (Mat_<double>(2, 1) << inlier_points_right_mat.at<double>(i, 0),
inlier_points_right_mat.at<double>(i, 1));
Mat right_reprjected = (Mat_<double>(2, 1) << reprojected_points.at<double>(i, 0),
reprojected_points.at<double>(i, 1));
double re_error = norm(right - right_reprjected);

///Check if points are within bound
if (inlier_points_left_mat.at<double>(i, 0) < x_upper &&
inlier_points_left_mat.at<double>(i, 0) > x_lower &&
inlier_points_left_mat.at<double>(i, 1) < y_upper &&
inlier_points_left_mat.at<double>(i, 1) > y_lower &&
inlier_points_right_mat.at<double>(i, 0) < x_upper &&
inlier_points_right_mat.at<double>(i, 0) > x_lower &&
inlier_points_right_mat.at<double>(i, 1) < y_upper &&
inlier_points_right_mat.at<double>(i, 1) > y_lower)
{

//Add 3D point to our global point cloud
CloudPoint newpoint;
newpoint.pt = Point3d(points3D.at<double>(i, 0), points3D.at<double>(i, 1),
points3D.at<double>(i, 2));
newpoint.frame_point.insert(pair<int, int>(a, refined_matches[i].trainIdx));
newpoint.reprojection_error = re_error;
pcloud.push_back(newpoint);

//Add 3D point to pcl visualizer
pcl::PointXYZ pclp; // Convert 3D point type to PCL type
pclp.x = points3D.at<double>(i, 0);
pclp.y = points3D.at<double>(i, 1);
pclp.z = points3D.at<double>(i, 2);
cloud->push_back(pclp);

//Push back the reprojection error
reproject_error_bound.push_back(re_error);

//Increase the number of points in the bound
in_bound++;
}

//Push back the reprojection error - as if every point was triangulated
reproject_error.push_back(re_error);
```

```
}
}

//Record the reconstruction time of this frame
double recon_time = difftime(time(0), frame_time);

//Record the total time of the algorithm so far
algorithmtime = difftime(time(0), start);

//Work out the mean and stdev reprojection error
double sum_reprojection = accumulate(begin(reproject_error), end(reproject_error), 0.0);
double mean_reprojection = sum_reprojection / reproject_error.size();

double accum_reprojection = 0.0;
for_each(begin(reproject_error), end(reproject_error), [&](const double d)
{
accum_reprojection += (d - mean_reprojection) * (d - mean_reprojection); //functor
}
);

// populate stdev
double stdev_reprojection = sqrt(accum_reprojection / (reproject_error.size() ));


//Work out the mean and stdev reprojection error in the bound

double sum_reprojection_bound = accumulate(begin(reproject_error_bound),
end(reproject_error_bound), 0.0);
double mean_reprojection_bound = sum_reprojection_bound / reproject_error_bound.size();

double accum_reprojection_bound = 0.0;
for_each(begin(reproject_error_bound), end(reproject_error_bound), [&](const double d)
{
accum_reprojection_bound += (d - mean_reprojection_bound) *
(d - mean_reprojection_bound); //functor
}
);

double stdev_reprojection_bound = sqrt(accum_reprojection_bound /
(reproject_error_bound.size() ));

//Add the mean reprojection error to the reprojection error
//vector of all mean reprojection errors
projection_error.push_back(mean_reprojection);
projection_error_bound.push_back(mean_reprojection_bound);

//Show the cloud

viewer.showCloud(cloud);

//Output to screen
cout << "--------------------------------------------" << endl;
cout << "--------------------------------------------" << endl;
cout << "Frame number: " << a << endl;
cout << " " << endl;
cout << "Number of keypoints: " << keypoints_left.size() << " " <<
keypoints_right.size() << endl;
cout << " " << endl;
cout << "Number of matches: " << size_of_matches << " " << endl;
cout << " " << endl;
cout << "Number of reconstruction points (refined matches): " <<
size_of_refinedmatches << endl;
cout << " " << endl;
cout << "Number of reconstruction points (within bound ): " << in_bound << endl;
cout << " " << endl;
cout << "Reprojection error : " << mean_reprojection << " " << stdev_reprojection << endl;
cout << " " << endl;
cout << "Reprojection error in bound: " << mean_reprojection_bound << " "
<< stdev_reprojection_bound << endl;
cout << " " << endl;
cout << "Point cloud size: " << pcloud.size() << endl;
cout << " " << endl;
cout << "Normalized camera matrix (Extrinsic parameters) " << endl;
cout << cameraextrinsic_right << endl;
```

```
cout << " " << endl;
cout << "Camera position: " << endl;
cout << camera_position << endl;
cout << " " << endl;
cout << "Camera orientation " << endl;
cout << camera_orientation << endl;
cout << " " << endl;
cout << "Camera orientation (euler angles)" << endl;
cout << camera_euler_orientation.t() << endl;
cout << " " << endl;
cout << "Time of frame reconstruction " << recon_time << endl;
cout << " " << endl;
cout << "Total time " << algorithmtime << endl;
cout << " " << endl;
cout << "-------------------------------------------------------------" << endl;
cout << "-------------------------------------------------------------" << endl;

//Output to log file
openfile << "-------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------" << endl;
openfile << "Frame number: " << a << endl;
openfile << " " << endl;
openfile << "Number of keypoints: " << keypoints_left.size() << " "
<< keypoints_right.size() << endl;
openfile << " " << endl;
openfile << "Number of matches: " << size_of_matches << " " << endl;
openfile << " " << endl;
openfile << "Number of reconstruction points (refined matches): "
<< size_of_refinedmatches << endl;
openfile << " " << endl;
openfile << "Number of reconstruction points (within bound ): " << in_bound << endl;
openfile << " " << endl;
openfile << "Reprojection error : " << mean_reprojection << " "
<< stdev_reprojection << endl;
openfile << " " << endl;
openfile << "Reprojection error in bound: " << mean_reprojection_bound
<< " " << stdev_reprojection_bound << endl;
openfile << " " << endl;
openfile << "Point cloud size: " << pcloud.size() << endl;
openfile << " " << endl;
openfile << "Normalized camera matrix (Extrinsic parameters) " << endl;
openfile << cameraextrinsic_right << endl;
openfile << " " << endl;
openfile << "Camera position: " << endl;
openfile << camera_position << endl;
openfile << " " << endl;
openfile << "Camera orientation " << endl;
openfile << camera_orientation << endl;
openfile << " " << endl;
openfile << "Camera orientation (euler angles)" << endl;
openfile << camera_euler_orientation.t() << endl;
openfile << " " << endl;
openfile << "Time of frame reconstruction " << recon_time << endl;
openfile << " " << endl;
openfile << "Total time " << algorithmtime << endl;
openfile << " " << endl;
openfile << "-------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------" << endl;

}
//Increase frame number
a++;
}

catch (...)
{
//This is incase the vector of frames has nothing in it,
//because we are running the processing and grab frame threads at the same time
}

}

cout << "Processing completed" << endl;
```

```
//----------PRINT OUT THE RESULTS OF RECONSTRUCTION AND SAVE TO FILE----------



//-------------------------------------------------------------
//Reprojection errors
//-------------------------------------------------------------

cout << "-------------------------------------------------------------" << endl;
cout << "-------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------" << endl;

cout << "Mean Reprojection errors" << endl;
openfile << "Mean Reprojection errors" << endl;
for (int i = 0; i < projection_error.size(); i++)
{
cout << projection_error[i] << " ";
openfile << projection_error[i] << " ";
}

//Work out the mean and stdev reprojection of reprojection errors in bound
double sum_projection_error = accumulate(begin(projection_error), end(projection_error), 0.0);
double mean_projection_error = sum_projection_error / projection_error.size();

double accum_projection_error = 0.0;
for_each(begin(projection_error), end(projection_error), [&](const double d)
{
accum_projection_error += (d - mean_projection_error) * (d - mean_projection_error); //functor
}
);

double stdev_projection_error = sqrt(accum_projection_error / (projection_error.size()));

cout << "Overall Mean reprojection error " << mean_projection_error << endl;
openfile << "Overall Mean reprojection error " << mean_projection_error << endl;

cout << "Overall Stdev reprojection error " << stdev_projection_error << endl;
openfile << "Overall Stdev reprojection error " << stdev_projection_error << endl;

//-------------------------------------------------------------
//Reprojection errors in bound
//-------------------------------------------------------------

cout << "-------------------------------------------------" << endl;
cout << "-------------------------------------------------" << endl;
openfile << "-------------------------------------------------" << endl;
openfile << "-------------------------------------------------" << endl;

cout << "Reprojection errors in bound" << endl;
openfile << "Reprojection errors in bound" << endl;
for (int i = 0; i < projection_error_bound.size(); i++)
{
cout << projection_error_bound[i] << " ";
openfile << projection_error_bound[i] << " ";
}

//Work out the mean and stdev reprojection of reprojection errors in bound
double sum_projection_error_bound = accumulate(begin(projection_error_bound),
end(projection_error_bound), 0.0);
double mean_projection_error_bound = sum_projection_error_bound /
projection_error_bound.size();

double accum_projection_error_bound = 0.0;
for_each(begin(projection_error_bound), end(projection_error_bound), [&](const double d)
{
accum_projection_error_bound += (d - mean_projection_error_bound) *
(d - mean_projection_error_bound); //functor
}
);

double stdev_projection_error_bound = sqrt(accum_projection_error_bound /
(projection_error_bound.size() ));
```

```
cout << "Mean reprojection error in bound " << mean_projection_error_bound << endl;
openfile << "Mean reprojection error in bound " << mean_projection_error_bound << endl;

cout << "Stdev reprojection error in bound " << stdev_projection_error_bound << endl;
openfile << "Stdev reprojection error in bound " << stdev_projection_error_bound << endl;

//-------------------------------------------------------------------------
//True Camera Orientations
//-------------------------------------------------------------------------

cout << "-------------------------------------------------------" << endl;
cout << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;

cout << "True Camera Orientations" << endl;
openfile << "True Camera Orientations" << endl;
for (int i = 0; i < camera_orientations_true.size(); i++)
{
cout << camera_orientations_true[i][0] << " " << camera_orientations_true[i][1] << " "
<< camera_orientations_true[i][2] << endl;
openfile << camera_orientations_true[i][0] << " " << camera_orientations_true[i][1] << " "
<< camera_orientations_true[i][2] << endl;
}


//-----------------------------------------------------
//Measured Camera Orientations
//-----------------------------------------------------

cout << "-------------------------------------------------------" << endl;
cout << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;

cout << "Measured Camera Orientations" << endl;
openfile << "Measured Camera Orientations" << endl;
for (int i = 0; i < camera_orientations_measured.size(); i++)
{
cout << camera_orientations_measured[i].at<double>(0) << " "
<< camera_orientations_measured[i].at<double>(1) << " "
<< camera_orientations_measured[i].at<double>(2) << endl;
openfile << camera_orientations_measured[i].at<double>(0) << " "
<< camera_orientations_measured[i].at<double>(1) << " "
<< camera_orientations_measured[i].at<double>(2) << endl;
}


//-------------------------------------------------------------------------------------
//Camera Orientation errors
//-------------------------------------------------------------------------------------

cout << "-------------------------------------------------------" << endl;
cout << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;

cout << "Camera Orientation Errors" << endl;
openfile << "Camera Orientation Errors" << endl;


for (int i = 0; i < camera_orientations_error.size(); i++)
{
cout << camera_orientations_error[i].at<double>(0) << " "
<< camera_orientations_error[i].at<double>(1) << " "
<< camera_orientations_error[i].at<double>(2) << endl;
openfile << camera_orientations_error[i].at<double>(0) << " "
<< camera_orientations_error[i].at<double>(1) << " "
<< camera_orientations_error[i].at<double>(2) << endl;
}


//Work out the mean and stdev reprojection of orientation errors wrt x y z

//---x
```

```
double sum_camera_orientations_x_error = accumulate(begin(camera_orientations_x_error),
end(camera_orientations_x_error), 0.0);
double mean_camera_orientations_x_error = sum_camera_orientations_x_error /
camera_orientations_x_error.size();

double accum_camera_orientations_x_error = 0.0;
for_each(begin(camera_orientations_x_error), end(camera_orientations_x_error), [&](const double d)
{
accum_camera_orientations_x_error += (d - mean_camera_orientations_x_error) *
(d - mean_camera_orientations_x_error); //functor
}
);

double stdev_camera_orientations_x_error = sqrt(accum_camera_orientations_x_error
/ (camera_orientations_x_error.size() ));

//---y
double sum_camera_orientations_y_error = accumulate(begin(camera_orientations_y_error),
end(camera_orientations_y_error), 0.0);
double mean_camera_orientations_y_error = sum_camera_orientations_y_error /
camera_orientations_y_error.size();

double accum_camera_orientations_y_error = 0.0;
for_each(begin(camera_orientations_y_error), end(camera_orientations_y_error), [&](const double d)
{
accum_camera_orientations_y_error += (d - mean_camera_orientations_y_error) *
(d - mean_camera_orientations_y_error); //functor
}
);

double stdev_camera_orientations_y_error = sqrt(accum_camera_orientations_y_error /
(camera_orientations_y_error.size() ));

//---z
double sum_camera_orientations_z_error = accumulate(begin(camera_orientations_z_error),
end(camera_orientations_z_error), 0.0);
double mean_camera_orientations_z_error = sum_camera_orientations_z_error /
camera_orientations_z_error.size();

double accum_camera_orientations_z_error = 0.0;
for_each(begin(camera_orientations_z_error), end(camera_orientations_z_error),
[&](const double d)
{
accum_camera_orientations_z_error += (d - mean_camera_orientations_z_error) *
(d - mean_camera_orientations_z_error); //functor
}
);

double stdev_camera_orientations_z_error = sqrt(accum_camera_orientations_z_error /
(camera_orientations_z_error.size() ));

cout << "Mean Camera Orientation Error X,Y,Z " << mean_camera_orientations_x_error
<< " " << mean_camera_orientations_y_error << " " << mean_camera_orientations_z_error << endl;
openfile << "Mean Camera Orientation Error X,Y,Z " << mean_camera_orientations_x_error << " "
<< mean_camera_orientations_y_error << " " << mean_camera_orientations_z_error << endl;

cout << "Stddev Camera Orientation Error X,Y,Z " << stdev_camera_orientations_x_error
<< " " << stdev_camera_orientations_y_error << " " << stdev_camera_orientations_z_error << endl;
openfile << "Stddev Camera Orientation Error X,Y,Z " << stdev_camera_orientations_x_error
<< " " << stdev_camera_orientations_y_error << " " << stdev_camera_orientations_z_error << endl;

//-------------------------------------------------------------
//True Camera Positions
//-------------------------------------------------------------

cout << "-------------------------------------------------------" << endl;
cout << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------" << endl;

cout << "True Camera Positions" << endl;
openfile << "True Camera Positions" << endl;
for (int i = 0; i < camera_positions_true.size(); i++)
{
cout << camera_positions_true[i][0] << " " << camera_positions_true[i][1]
```

```
<< " " << camera_positions_true[i][2] << endl;
openfile << camera_positions_true[i][0] << " " << camera_positions_true[i][1]
<< " " << camera_positions_true[i][2] << endl;
}


//-------------------------------------------------------------------
//Measured Camera Positions
//-------------------------------------------------------------------

cout << "-------------------------------------------------------------------" << endl;
cout << "-------------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------------" << endl;

cout << "Measured Camera Positions" << endl;
openfile << "Measured Camera Positions" << endl;
for (int i = 0; i < camera_positions_measured.size(); i++)
{
cout << camera_positions_measured[i].at<double>(0) << " "
<< camera_positions_measured[i].at<double>(1) << " "
<< camera_positions_measured[i].at<double>(2) << endl;
openfile << camera_positions_measured[i].at<double>(0) << " "
<< camera_positions_measured[i].at<double>(1) << " "
<< camera_positions_measured[i].at<double>(2) << endl;
}

//-------------------------------------------------------------------
//Camera Position Errors
//-------------------------------------------------------------------

cout << "-------------------------------------------------------------------" << endl;
cout << "-------------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------------" << endl;
openfile << "-------------------------------------------------------------------" << endl;

cout << "Camera Positions Errors" << endl;
openfile << "Camera Positions Errors" << endl;

for (int i = 0; i < camera_positions_error.size(); i++)
{
cout << camera_positions_error[i] << endl;
openfile << camera_positions_error[i] << endl;

}

//Work out the mean and stdev reprojection of positions errors

double sum_camera_positions_error = accumulate(begin(camera_positions_error),
end(camera_positions_error), 0.0);
double mean_camera_positions_error = sum_camera_positions_error / camera_positions_error.size();

double accum_camera_positions_error = 0.0;
for_each(begin(camera_positions_error), end(camera_positions_error), [&](const double d)
{
accum_camera_positions_error += (d - mean_camera_positions_error) *
(d - mean_camera_positions_error); //functor
}
);

double stdev_camera_positions_error = sqrt(accum_camera_positions_error
/ (camera_positions_error.size() ));

//----------


cout << "RECONSTRUCTION RP3DRv10 " << string(keypoint_detection) << " "
<< string(keypoint_matching) << " " << string(ego_motion)
<< " O/Os/P/Ps/time/size/Rb/Rbs/R/Rs " << mean_camera_orientations_x_error << " "
<< mean_camera_orientations_y_error << " " << mean_camera_orientations_z_error
<< " " << stdev_camera_orientations_x_error << " " << stdev_camera_orientations_y_error
<< " " << stdev_camera_orientations_z_error <<" "<< mean_camera_positions_error << " "
<< stdev_camera_positions_error << " " << algorithmtime << " " << pcloud.size() << " "
<< mean_projection_error_bound << " " << stdev_projection_error_bound << " "
<< mean_projection_error << " " << stdev_projection_error << endl;
```

```
openfile << "RECONSTRUCTION RP3DRv10 " << string(keypoint_detection)
<< " " << string(keypoint_matching) << " " << string(ego_motion)
<< " O/Os/P/Ps/time/size/Rb/Rbs/R/Rs " << mean_camera_orientations_x_error
<< " " << mean_camera_orientations_y_error << " "
<< mean_camera_orientations_z_error << " " << stdev_camera_orientations_x_error
<< " " << stdev_camera_orientations_y_error << " " << stdev_camera_orientations_z_error
<< " " << mean_camera_positions_error << " " << stdev_camera_positions_error << " "
<< algorithmtime << " " << pcloud.size() << " " << mean_projection_error_bound << " "
<< stdev_projection_error_bound << " " << mean_projection_error << " "
<< stdev_projection_error << endl;
//Save point cloud struct to file

for (int i = 0; i < pcloud.size(); i++)
{
openfile2 << pcloud[i].pt.x << " " << pcloud[i].pt.y << " " << pcloud[i].pt.z << " "
<< pcloud[i].reprojection_error << " " << endl;
}

//Clear the vectors
camera_positions_measured.clear();
camera_orientations_measured.clear();
camera_positions_error.clear();
camera_orientations_error.clear();
camera_orientations_x_error.clear();
camera_orientations_y_error.clear();
camera_orientations_z_error.clear();
camera_positions_error.clear();
projection_error.clear();
projection_error_bound.clear();

}

//Main function
int main(int argc, char** argv)
{
cout << "Welcome to Real-time Progressive Stereo Reconstruction RP3DR V10 " << endl;
cout << " " << endl;

Mat_<double> KMatrix; //Calibration matrix
Mat_<double> distortion_coeff; //Distortion coefficients
tbb::concurrent_vector<cv::Mat> frames; //Frame vector
string keypoint_detection, keypoint_matching, ego_motion; //The algorithms used

string frameaquisitiondecision;
cout << "Enter DATASET to use the DATASET or REALTIME for the real-time application" << endl;
cin >> framcaquisitiondecision;

if (frameaquisitiondecision == "DATASET")
{
//Get frames from file
string filename = "Fountain/01.png";

string batch;
cout << "Enter BATCH if you would like to batch process all best practice combinations ";
cout << "or enter any key if you would like to define your own best practice combination" << endl;
cin >> batch;

if (batch == "BATCH")
{
vector<vector<string>> combinations{
{ "AKAZE", "opticalflow", "8point" },
{ "AKAZE", "opticalflow", "7point" },
{ "AKAZE", "opticalflow", "5point" },
{ "AKAZE", "featurebrutebinary", "8point" },
{ "AKAZE", "featurebrutebinary", "7point" },
{ "AKAZE", "featurebrutebinary", "5point" },
{ "BRISK", "opticalflow", "8point" },
{ "BRISK", "opticalflow", "7point" },
{ "BRISK", "opticalflow", "5point" },
{ "BRISK", "featurebrutebinary", "8point" },
{ "BRISK", "featurebrutebinary", "7point" },
{ "BRISK", "featurebrutebinary", "5point" },
{ "SIFT", "opticalflow", "8point" },
{ "SIFT", "opticalflow", "7point" },
{ "SIFT", "opticalflow", "5point" },
```

```
{ "SIFT", "featurebrute", "8point" },
{ "SIFT", "featurebrute", "7point" },
{ "SIFT", "featurebrute", "5point" },
{ "SIFT", "featureapprox", "8point" },
{ "SIFT", "featureapprox", "7point" },
{ "SIFT", "featureapprox", "5point" },
{ "SURF", "opticalflow", "8point" },
{ "SURF", "opticalflow", "7point" },
{ "SURF", "opticalflow", "5point" },
{ "SURF", "featurebrute", "8point" },
{ "SURF", "featurebrute", "7point" },
{ "SURF", "featurebrute", "5point" },
{ "SURF", "featureapprox", "8point" },
{ "SURF", "featureapprox", "7point" },
{ "SURF", "featureapprox", "5point" }
};

//Set calibration matrix
KMatrix = (Mat_<double>(3, 3) << 2759.48, 0, 1520.69,    0, 2764.16, 1006.81,
0, 0, 1);
distortion_coeff = cv::Mat_<double>::zeros(1, 4);

for (int i = 0; i < combinations.size(); i++)
{
vector<string> combination = combinations[i];
keypoint_detection = combination[0];
keypoint_matching = combination[1];
ego_motion = combination[2];
time_t start = time(0);

//Run both functions filestream and process at same time
boost::thread t1(&Filestream, filename, boost::ref(frames));
boost::thread t2(&Stream_process, KMatrix, distortion_coeff, boost::ref(frames),
keypoint_detection, keypoint_matching, ego_motion, start);
t1.join(); //Wait for the thread to finish
t2.join(); //Wait for thread to finish
frames.clear();
}
}

else
{
cout << "Enter the keypoint detection algorithm SIFT/SURF/FAST/HOUGH" << endl;
cin >> keypoint_detection;

cout << "Enter the keypoint matching algorithm opticalflow/featureapprox/featurebrute" << endl;
cin >> keypoint_matching;

cout << "Enter the ego_motion algorithm 5point/7point/8point," << endl;
cin >> ego_motion;

//Set calibration matrix
KMatrix = (Mat_<double>(3, 3) << 2759.48, 0, 1520.69,
0, 2764.16, 1006.81,
0, 0, 1);
distortion_coeff = cv::Mat_<double>::zeros(1, 4);

//Begin timing
time_t start = time(0);

//Run both functions filestream and process at same time
boost::thread t1(&Filestream, filename, boost::ref(frames));
boost::thread t2(&Stream_process, KMatrix, distortion_coeff, boost::ref(frames),
keypoint_detection, keypoint_matching, ego_motion, start);
t1.join(); //Wait for the thread to finish
t2.join(); //Wait for thread to finish

}

}

else if (frameaquisitiondecision == "REALTIME")
{
cout << "Enter the keypoint detection algorithm SIFT/SURF/FAST/HOUGH" << endl;
cin >> keypoint_detection;
```

```
cout << "Enter the keypoint matching algorithm opticalflow/featureapprox/featurebrute" << endl;
cin >> keypoint_matching;

cout << "Enter the ego_motion algorithm 5point/7point/8point," << endl;
cin >> ego_motion;

int stream_code;
cout << "Enter stream_code" << endl;
cin >> stream_code;

//Calibrate camera
boost::tie(KMatrix, distortion_coeff) = Stream_Calibrate(stream_code);

//Begin timing
time_t start = time(0);

boost::thread t1(&Livestream, stream_code, boost::ref(frames));
boost::thread t2(&Stream_process, KMatrix, distortion_coeff, boost::ref(frames),
keypoint_detection, keypoint_matching, ego_motion, start);
t1.join(); //Wait for the thread to finish
t2.join(); //Wait for thread to finish
}



//Pause the program
system("pause");
return 0;
};
```

LISTING D.1: Main Class

D.2 Optical Flow Class

tabsize

```
//Real-time Progressive Stereo Reconstruction
//RP3DR V7
//25 September
//Matthew Westaway
//Optical Flow Class


//Headerfile includes
#pragma once //Ensure it hasnt been included before
#include "opticalflow.h"

//Constructor
Opticalflow::Opticalflow(Mat frame1, Mat frame2)
{
img1 = frame1;
img2 = frame2;
}
//Function to get matches by optical flow
vector<DMatch> Opticalflow::Getmatches(vector<KeyPoint> keypoints1, vector<KeyPoint> keypoints2)
{
//Convert Keypoints in left image to points
vector<Point2f> keypoints_left;

for (int i = 0; i<keypoints1.size(); i++)
{
keypoints_left.push_back(keypoints1[i].pt);
}

//Convert Keypoints in right image to points
vector<Point2f> keypoints_right;
for (int i = 0; i<keypoints2.size(); i++)
{
keypoints_right.push_back(keypoints2[i].pt);
}

//Create points in the right image - this will be location
//containing the calculated new positions of left points in the second image
vector<Point2f> optical_points_right;

//Convert images to grey scale
Mat img1_grey, img2_grey;
cvtColor(img1, img1_grey, CV_RGB2GRAY);
cvtColor(img2, img2_grey, CV_RGB2GRAY);

//Calculate the optical flow field - how each left point moves accross the two images
vector<uchar> optical_status; //1 if flow has been found for feature
vector<float> optical_errors;
calcOpticalFlowPyrLK(img1_grey, img2_grey, keypoints_left,
optical_points_right, optical_status, optical_errors);

//Filter out the points with high error
vector<Point2f>optical_points_right_good;
vector<int> optical_points_right_good_index;

for (int i = 0; i<optical_status.size(); i++)
{
//If point is an inlier and has a low error,whats vstatus[i]
if (optical_status[i] && optical_status[i]<12)

{
//Keep the original index of the point
optical_points_right_good_index.push_back(i);

//Keep the point itself
optical_points_right_good.push_back(optical_points_right[i]);
}
else
```

```
{
optical_status[i] = 0;//a bad flow
}
}

//Look round each optical flow point in the right image for
//keypoints and see if can make a match.
//First reshape keypoints to 1 channel
Mat optical_points_right_good_flat = Mat(optical_points_right_good).reshape(1,
optical_points_right_good.size());
Mat keypoints_right_flat = Mat(keypoints_right).reshape(1, keypoints_right.size());

vector<vector<DMatch>> nearest_neighbours; //the matches
vector<DMatch> matches;

BFMatcher matcher(NORM_L2, true);
matcher.radiusMatch(optical_points_right_good_flat,
keypoints_right_flat, nearest_neighbours, 2.0f); //Max distance is 2


//Check that the found neighbors are unique
//(through away neighbours that are too close together, as they may be confusing)
set <int> found_keypoints_right; //for duplicate prevention

//Go through each match and look at how many neighbours it has
for (int i = 0; i<nearest_neighbours.size(); i++)

{
//Find the matching point
DMatch matching_keypoint;

if (nearest_neighbours[i].size() == 1)
{
//Only one neighbours
matching_keypoint = nearest_neighbours[i][0];
}

else if (nearest_neighbours[i].size()>1)
{
//2 neighbours - check how close they are
double ratio = nearest_neighbours[i][0].distance
/ nearest_neighbours[i][1].distance;

if (ratio <0.7) //Not too close
{
//take the first neighbours
matching_keypoint = nearest_neighbours[i][0];
}
else
{
//Neighbouring points are too close we cant tell which point is better -
//throw away the point
continue; //did not pass ratio test
}

}
else //no neighbours
{
continue;
}

//If the matched point we have now has not been matched before i.e
//it cannot be found in the index of matched points
if (found_keypoints_right.find(matching_keypoint.trainIdx)
== found_keypoints_right.end())
{
//The found neighbour was not yet used - match with original indexing
//instead of index of optical flow point
matching_keypoint.queryIdx =
optical_points_right_good_index[matching_keypoint.queryIdx];
matches.push_back(matching_keypoint);//add this match
found_keypoints_right.insert(matching_keypoint.trainIdx);
}
}
```

```
return matches;
}
```

LISTING D.2: Optical Flow Class

Bibliography

Aeschliman, C. (2008), 'Ece661 computer vision, homework 2', University Lecture.

Ahmed, M. T., Dailey, M. N., Landabaso, J. L. and Herrero, N. (2010), Robust key frame extraction for 3d reconstruction from video streams., *in* 'VISAPP (1)', pp. 231–236.

Bao, S. Y., Furlan, A., Fei-Fei, L. and Savarese, S. (2014), Understanding the 3d layout of a cluttered room from multiple images, *in* 'Applications of Computer Vision (WACV), 2014 IEEE Winter Conference on', IEEE, pp. 690–697.

Brilakis, I., Fathi, H. and Rashidi, A. (2011), 'Progressive 3d reconstruction of infrastructure with videogrammetry', *Automation in Construction* **20**(7), 884–895.

Collins, R. (2007), 'Cse/ee486 computer vision i, lecture notes', University Lecture.

Douxchamps, D. and Macq, B. (2004), Integrating perspective distortions in stereo image matching, *in* 'Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on', Vol. 3, IEEE, pp. iii–301.

Geiger, A., Ziegler, J. and Stiller, C. (2011), Stereoscan: Dense 3d reconstruction in real-time, *in* 'Intelligent Vehicles Symposium (IV), 2011 IEEE', IEEE, pp. 963–968.

Hartley, R. and Zisserman, A. (2003), *Multiple view geometry in computer vision*, Cambridge university press.

Kang, Z. and Medioni, G. (2015), Progressive 3d model acquisition with a commodity hand-held camera, *in* 'Applications of Computer Vision (WACV), 2015 IEEE Winter Conference on', IEEE, pp. 270–277.

Medioni, G. and Kang, S. B. (2004), *Emerging topics in computer vision*, Prentice Hall PTR.

Mouragnon, E., Lhuillier, M., Dhome, M., Dekeyser, F. and Sayd, P. (2006), Real time localization and 3d reconstruction, *in* 'Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on', Vol. 1, IEEE, pp. 363–370.

Nistér, D. (2004), 'An efficient solution to the five-point relative pose problem', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **26**(6), 756–770.

Owens, R. (1997), 'Computer vision it412, lecture 11 stereo matching', University Lecture.

Peyré, G. (2012), 'Introduction to images', University Lecture.

Pollefeys, M., Koch, R., Vergauwen, M. and Van Gool, L. (1999), Hand-held acquisition of 3d models with a video camera, *in* '3-D Digital Imaging and Modeling, 1999. Proceedings. Second International Conference on', IEEE, pp. 14–23.

Pradeep, V., Rhemann, C., Izadi, S., Zach, C., Bleyer, M. and Bathiche, S. (2013), Monofusion: Real-time 3d reconstruction of small scenes with a single web camera, *in* 'Mixed and Augmented Reality (ISMAR), 2013 IEEE International Symposium on', IEEE, pp. 83–88.

Rashidi, A., Dai, F., Brilakis, I. and Vela, P. (2013), 'Optimized selection of key frames for monocular videogrammetric surveying of civil infrastructure', *Advanced Engineering Informatics* **27**(2), 270–282.

Se, S. and Jasiobedzki, P. (2006), Photo-realistic 3d model reconstruction, *in* 'Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on', IEEE, pp. 3076–3082.

Strecha, C., von Hansen, W., Gool, L. V., Fua, P. and Thoennessen, U. (2008), On benchmarking camera calibration and multi-view stereo for high resolution imagery, *in* 'Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on', IEEE, pp. 1–8.

Teng, C.-H. (2014), 'Improving three-dimensional point reconstruction from image correspondences using surface curvatures', *Machine vision and applications* **25**(2), 421–436.

Wang, C., Bao, M. and Shen, T. (2012), '3d model reconstruction algorithm and implementation based on the mobile device 1'.

Watson, M., Bhatti, A., Abdi, H. and Nahavandi, S. (2012), *Generation of 3D sparse feature models using multiple stereo views*, InTech.

Wikipedia (2015*a*), 'Active contour model — Wikipedia, the free encyclopedia', `http://en.wikipedia.org/w/index.php?title=Active%20contour%20model&oldid=678410704`.

Wikipedia (2015*b*), 'Hough transform — Wikipedia, the free encyclopedia', `http://en.wikipedia.org/w/index.php?title=Hough%20transform&oldid=679388059`.

Xiang, Z. and Sheng-yong, C. (2011), 'A 3d image reconstruction model from multiple images', *Procedia Engineering* **23**, 678–683.

Zakharov, A. and Barinov, A. (2015), 'An algorithm for 3d-object reconstruction from video using stereo correspondences', *Pattern Recognition and Image Analysis* **25**(1), 117–121.